

conference

proceedings

2001 USENIX Annual Technical Conference

Boston, Massachusetts, USA

June 25–30, 2001

Sponsored by
The USENIX Association



For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: <http://www.usenix.org>

The price is \$32 for members and \$40 for nonmembers.

Outside the U.S.A. and Canada, please add
\$18 per copy for postage (via air printed matter).

Past USENIX Technical Conferences

2000 San Diego, CA	1990 Winter Washington, D.C.
1999 Monterey CA	1989 Summer Baltimore
1998 New Orleans	1989 Winter San Diego
1997 Anaheim	1988 Summer San Francisco
1996 San Diego	1988 Winter Dallas
1995 New Orleans	1987 Summer Phoenix
1994 Summer Boston	1987 Winter Washington, D.C.
1994 Winter San Francisco	1986 Summer Atlanta
1993 Summer Cincinnati	1986 Winter Denver
1993 Winter San Diego	1985 Summer Portland
1992 Summer San Antonio	1985 Winter Dallas
1992 Winter San Francisco	1984 Summer Salt Lake City
1991 Summer Nashville	1984 Winter Washington, D.C.
1991 Winter Dallas	1983 Summer Toronto
1990 Summer Anaheim	1983 Winter San Diego

© 2001 by The USENIX Association
All Rights Reserved

This volume is published as a collective work. Rights to individual papers remain with the author or the author's employer. Permission is granted for the noncommercial reproduction of the complete work for educational or research purposes. USENIX acknowledges all trademarks herein.

ISBN 1-880446-09-X

Printed in the United States of America on 50% recycled paper, 10–15% post consumer waste.

USENIX Association

**Proceedings of the
2001 USENIX
Annual Technical Conference**

**June 25–30, 2001
Boston, Massachusetts, USA**

2001 USENIX Annual Technical Conference

June 25–30, 2001
Boston, Massachusetts, USA

Index of Authorsvii
Message from the Program Chairix

Thursday, June 28

Operating Systems

Session Chair: Jochen Liedtke, University of Karlsruhe

Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor	1
<i>Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim, VMware Inc.</i>	

Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources	15
<i>Jeff Bonwick, Sun Microsystems, and Jonathan Adams, California Institute of Technology</i>	

Measuring Thin-Client Performance Using Slow-Motion Benchmarking	35
<i>S. Jae Yang, Jason Nieh, and Naomi Novik, Columbia University</i>	

Security

Session Chair: Dan Wallach, Rice University

An Architecture for Secure Generation and Verification of Electronic Coupons	51
<i>Rahul Garg, Parul Mittal, Vikas Agarwal, and Natwar Modani, IBM India Research Lab</i>	

Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML	65
<i>Don Davis, Shym Technology</i>	

Unifying File System Protection	79
<i>Christopher A. Stein, Harvard University; John H. Howard, Sun Microsystems; and Margo I. Seltzer, Harvard University</i>	

Storage I

Session Chair: Greg Ganger, Carnegie Mellon University

The Multi-Queue Replacement Algorithm for Second Level Buffer Caches	91
<i>Yuanyuan Zhou, and James Philbin, NEC Research Institute; and Kai Li, Princeton University</i>	

Design and Implementation of a Predictive File Prefetching Algorithm	105
<i>Thomas M. Kroeger, Nokia Cluster IP Solutions; and Darrell D. E. Long, University of California, Santa Cruz</i>	

Extending Heterogeneity to RAID Level 5	119
<i>T. Cortes and J. Laborta, Universitat Politècnica de Catalunya</i>	

Friday, June 29

Tools

Session Chair: Wuchi Feng, Ohio State University

Reverse-Engineering Instruction Encodings	133
<i>Wilson C. Hsieh, University of Utah; Dawson Engler, Stanford University; and Godmar Back, University of Utah</i>	
An Embedded Error Recovery and Debugging Mechanism for Scripting Language Extensions	147
<i>David M. Beazley, University of Chicago</i>	
Interactive Simultaneous Editing of Multiple Text Regions	161
<i>Robert C. Miller and Brad A. Myers, Carnegie Mellon University</i>	

Web Servers

Session Chair: Mohit Aron, Zambeel Inc.

High-Performance Memory-Based Web Servers: Kernel and User-Space Performance	175
<i>Philippe Joubert, ReefEdge Inc.; Robert B. King, IBM Research; Richard Neves, ReefEdge Inc.; Mark Russinovich, Winternals Software; and John M. Tracey, IBM Research</i>	
Kernel Mechanisms for Service Differentiation in Overloaded Web Servers	189
<i>Thiemo Voigt, Swedish Institute of Computer Science; Renu Tewari and Douglas Freimuth, IBM T.J. Watson Research Center; and Ashish Mehra, iScale Networks</i>	
Storage Management for Web Proxies	203
<i>Elizabeth Shriver and Eran Gabber, Bell Labs; Lan Huang, SUNY Stony Brook; and Christopher A. Stein, Harvard University</i>	

Saturday, June 30

Scheduling

Session Chair: Sheila Harnett, IBM Linux Technology Center

Pragmatic Nonblocking Synchronization for Real-Time Systems	217
<i>Michael Hohmuth and Hermann Härtig, Dresden University of Technology</i>	
Scalability of Linux Event-Dispatch Mechanisms	231
<i>Abhishek Chandra, University of Massachusetts, Amherst; and David Mosberger, HP Labs</i>	
Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler	245
<i>Jason Nieh, Chris Vaill, and Hua Zhong, Columbia University</i>	

Storage II

Session Chair: Carla Ellis, Duke University

A Toolkit for User-Level File Systems	261
<i>David Mazières, NYU</i>	
Charm: An I/O-Driven Execution Strategy for High-Performance Transaction Processing	275
<i>Lan Huang and Tzi-cker Chiueh, State University of New York at Stony Brook</i>	
Fast Indexing: Support for Size-Changing Algorithms in Stackable File Systems	289
<i>Erez Zadok, SUNY Stony Brook; Johan M. Andersen, Ion Badulescu, and Jason Nieh, Columbia University</i>	

Networking

Session Chair: Robert Miller, Carnegie Mellon University

- Payload Caching: High-Speed Data Forwarding for Network Intermediaries 305
Kenneth Yocum and Jeffrey Chase, Duke University

- A Waypoint Service Approach to Connect Heterogeneous Internet Address Spaces 319
T. S. Eugene Ng, Ion Stoica, and Hui Zhang, Carnegie Mellon University

- Flexible Control of Parallelism in a Multiprocessor PC Router 333
Benjie Chen and Robert Morris, Massachusetts Institute of Technology

Index of Authors

Adams, Jonathan	15	Mehra, Ashish	189
Agarwal, Vikas	51	Miller, Robert C.	161
Andersen, Johan M.	289	Mittal, Parul	51
Back, Godmar	133	Modani, Natwar	51
Badulescu, Ion	289	Morris, Robert	333
Beazley, David M.	147	Mosberger, David	231
Bonwick, Jeff	15	Myers, Brad A.	161
Chase, Jeffrey	305	Neves, Richard	175
Chen, Benjie	333	Ng, T. S. Eugene	319
Chiueh, Tzi-cker	275	Nieh, Jason	35, 245, 289
Cortes, T.	119	Novik, Naomi	35
Chandra, Abhishek	231	Philbin, James	91
Davis, Don	65	Russinovich, Mark	175
Engler, Dawson	133	Seltzer, Margo I.	79
Freimuth, Douglas	189	Shriver, Elizabeth	203
Gabber, Eran	203	Stein, Christopher A.	79, 203
Garg, Rahul	51	Stoica, Ion	319
Härtig, Hermann	217	Sugerman, Jeremy	1
Hohmuth, Michael	217	Tewari, Renu	189
Howard, John H.	79	Tracey, John M.	175
Hsieh, Wilson C.	133	Vaill, Chris	245
Huang, Lan	203, 275	Venkitachalam, Ganesh	1
Joubert, Philippe	175	Voigt, Thiemo	189
King, Robert B.	175	Yang, S. Jae	35
Kroeger, Thomas M.	105	Yocum, Kenneth	305
Laborta, J.	119	Zadok, Erez	289
Li, Kai	91	Zhang, Hui	319
Lim, Beng-Hong	1	Zhong, Hua	245
Long, Darrell D. E.	105	Zhou, Yuanyuan	91
Mazières, David	261		

Message from the Conference Chair

Welcome to Boston and the 2001 USENIX Annual Technical Conference! This conference is the result of a lot of hard work, and many thanks are in order.

Clem Cole and the FREENIX program committee have put together an outstanding FREENIX Track. Clem's knowledge and experience have been invaluable to USENIX and to me in the past year. John Kohl and Matt Blaze have put together a great set of invited talks, providing attendees with the pleasant quandary of picking between invited talks and the technical tracks. Due to their popularity, three days of tutorials have again been scheduled. Of course, this is a testament to the hard work of the Tutorial Coordinator, Dan Klein.

The General Track program committee received 82 submissions. With the help of 35 external reviewers, the committee generated 354 reviews in five weeks, or slightly more than 4 reviews per paper. The committee was impressed with the quality of the submissions, which made choosing the 24 papers in the track a difficult but rewarding task. The committee members are very pleased with the quality and diversity of the 24 papers. We are sure you will agree that these papers continue the USENIX tradition of bridging the gap between researchers and developers. I cannot thank the committee and the reviewers enough for their efforts. Particular thanks are due to Peter Honeyman, who acted as my liaison to the USENIX Board. Two reviewers, Robert Gray and Jason Nieh, went well beyond the call of duty.

Thanks to Clem Cole, the committee members were able to meet in great comfort at the Compaq facilities in Littleton, Massachusetts. A very special thanks to the USENIX staff, especially Ellie Young and Jane-Ellen Long. The USENIX community is extremely fortunate to have both Ellie and Jane-Ellen making sure everything runs smoothly. Finally, I offer thanks to my former and current employers, IBM Watson and ReefEdge, for their patience and support.

Yoonho Park, Program Chair

Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor

Jeremy Sugerman, Ganesh Venkitachalam and Beng-Hong Lim
VMware, Inc.

3145 Porter Dr, Palo Alto, CA 94304
{yoel,ganesh,bhlim}@vmware.com

Abstract

Virtual machines were developed by IBM in the 1960's to provide concurrent, interactive access to a mainframe computer. Each virtual machine is a replica of the underlying physical machine and users are given the illusion of running directly on the physical machine. Virtual machines also provide benefits like isolation and resource sharing, and the ability to run multiple flavors and configurations of operating systems. VMware™Workstation brings such mainframe-class virtual machine technology to PC-based desktop and workstation computers.

This paper focuses on VMware Workstation's approach to virtualizing I/O devices. PCs have a staggering variety of hardware, and are usually pre-installed with an operating system. Instead of replacing the pre-installed OS, VMware Workstation uses it to host a user-level application (VMApp) component, as well as to schedule a privileged virtual machine monitor (VMM) component. The VMM directly provides high-performance CPU virtualization while the VMApp uses the host OS to virtualize I/O devices and shield the VMM from the variety of devices. A crucial question is whether virtualizing devices via such a hosted architecture can meet the performance required of high throughput, low latency devices.

To this end, this paper studies the virtualization and performance of an Ethernet adapter on VMware Workstation. Results indicate that with optimizations, VMware Workstation's hosted virtualization architecture can match native I/O throughput on standard PCs. Although a straightforward hosted implementation is CPU-limited due to virtualization overhead on a 733 MHz Pentium®III system on a 100 Mb/s Ethernet, a series of optimizations targeted at reducing CPU utilization allows the system to match native network throughput. Further optimizations are discussed both within and outside a hosted architecture.

1 Introduction

The concept of the virtual machine was invented by IBM as a method of time-sharing extremely expensive mainframe hardware [4, 5]. As defined by IBM, a "virtual machine" is

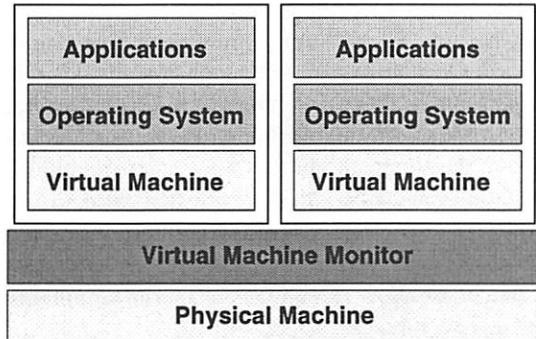


Figure 1: A virtual machine monitor provides a virtual machine abstraction in which standard operating systems and applications may run. Each virtual machine is fully isolated from the rest of the virtual machines.

a fully protected and isolated copy of the underlying physical machine's hardware. Thus, each virtual machine user is given the illusion of having a dedicated physical machine. Software developers can also write and test programs without fear of crashing the physical machine and affecting the other users.

Figure 1 illustrates the traditional organization of a virtual machine system. A software layer called a *virtual machine monitor (VMM)* takes complete control of the machine hardware and creates virtual machines, each of which behaves like a complete physical machine that can run its own operating system (OS). Contrast this with a normal system where a single operating system is in control of the machine.

To maximize performance, the monitor gets out of the way whenever possible and allows the virtual machine to execute directly on the hardware, albeit in a non-privileged mode. The monitor regains control whenever the virtual machine tries to perform an operation that may affect the correct operation of other virtual machines or of the hardware. The monitor safely emulates the operation before returning control to the virtual machine. This direct execution property allows mainframe-class virtual machines to achieve close to native performance and sets the technol-

ogy apart from machine emulators that always impose an extra layer of interpretation on the emulated machine.

The result of a complete machine virtualization is the creation of a set of virtual computers that runs on a physical computer. Different operating systems, or separate instances of the same operating system, can run in each virtual machine. The operating systems that run in virtual machines are termed *guest* operating systems. Since virtual machines are isolated from each other, a guest operating system crash does not affect the other virtual machines. Users in different virtual machines cannot affect each other catastrophically.

Most of the benefits of mainframe virtual machines apply to the PC platform, and several new ones have emerged. On mainframes, virtual machines have been used for timesharing, for partitioning machine resources among different OSes and applications, as well as for OS and software development and easing system migration. On a desktop or workstation PC there is a need to run different operating systems – primarily the various flavors of Microsoft® and UNIX™-based operating systems. Virtual machines allow these OSes to be run simultaneously on a single computer.

Intel®-based PCs are also increasingly being used as servers by traditional enterprises and service providers to host applications. Frequently, an entire machine is dedicated to a particular service, application or customer in order to provide fault isolation and performance guarantees. In this arena, virtual machines can be used to host applications, provide better resource utilization, and ease system manageability. Virtual machines can also be easily migrated and replicated across machines to aid in service provisioning. Virtual machines can contain identical virtual hardware, even on hosts with different native hardware, making virtual machines freely portable between different physical machines.

1.1 Virtualizing the PC platform

Several technical and pragmatic hurdles must be overcome when virtualizing the PC platform. The traditional mainframe approach runs virtual machines in a less privileged mode in order to allow the VMM to regain control on privileged instructions, and relies on the VMM to virtualize and interface directly to the I/O devices. Also, the VMM is in complete control of the entire machine. This approach doesn't apply as easily to PCs for the following reasons.

Non-virtualizable processor – The Intel IA-32 processor architecture [10] is not naturally virtualizable. Popek and Goldberg [11] showed that an architecture can support virtual machines only if all instructions that can inspect or modify privileged machine state will trap when executed from any but the most privileged mode. Because the IA-32 processor does not

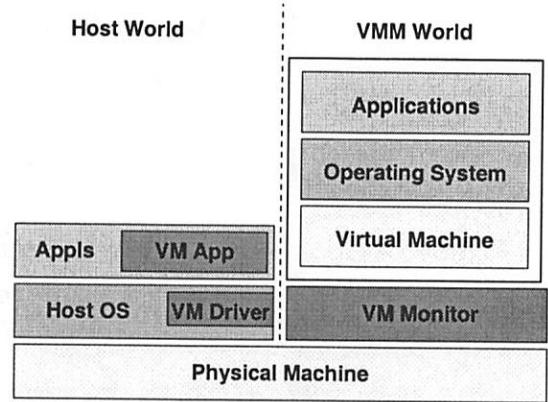


Figure 2: VMware's hosted virtual machine model splits the virtualization software between a virtual machine monitor that virtualizes the CPU, an application that uses a host operating system for device support, and an operating system driver for transitioning between them.

meet this condition, it is not possible to virtualize the processor by simply executing all virtual machine instructions in a less privileged mode.

PC hardware diversity – There is a large diversity of devices that may be found in PCs. This is a result of the PC's "open" architecture. In a traditional implementation, the virtual machine monitor would have to manage these devices. This would require a large programming effort to provide device drivers in the VMM for all supported PC devices.

Pre-existing PC software – Unlike mainframes that are configured and managed by experienced system administrators, desktop and workstation PC's are often pre-installed with a standard OS and set up and managed by the end-user. In this environment, it is extremely important to allow a user to adopt virtual machine technology without losing the ability to continue using his existing OS and applications. It would be unacceptable to completely replace an existing OS with a virtual machine monitor.

VMware Workstation has a *hosted* architecture that allows it to co-exist with a pre-existing *host* operating system, and rely upon that operating system for device support. Figure 2 illustrates the components of this hosted architecture. This architecture allows VMware to cope with the diversity of PC hardware and to be compatible with pre-existing PC software. Currently, Windows NT®, Windows®2000 and Linux can serve as hosts. This paper focuses on the performance aspects of relying on a host OS for accessing I/O devices.

The rest of this paper is organized as follows. Section 2 describes VMware Workstation's hosted architecture, its benefits and costs, and looks at the specific ex-

ample of a virtual Ethernet network interface card (NIC). Section 3 demonstrates the performance of NIC virtualization with VMware Workstation 2.0, breaks down the overheads for a few different workloads, and measures improvements achieved by optimizations the data suggested. Section 4 presents several approaches for improving I/O performance beyond the optimizations described in Section 3, some of which go beyond the capabilities of a hosted architecture. Section 5 describes related work in the area of supporting multi-platform computing on a single machine. Finally, Section 6 summarizes the observed properties of the hosted architecture and draws some conclusions about this approach to I/O virtualization.

2 A Hosted Virtual Machine Architecture

VMware Workstation virtualizes I/O devices using a novel design called the *Hosted Virtual Machine Architecture*. The primary feature of this design is that it takes advantage of a pre-existing operating system for I/O device support and still achieves near native performance for CPU-intensive workloads. Figure 2 illustrates the structure of a virtual machine in the hosted architecture.

VMware Workstation installs like a normal application on an operating system, known as the host operating system. When run, the application portion (**VMAApp**) uses a driver loaded into the host operating system (**VMDriver**) to establish the privileged virtual machine monitor component (**VMM**) that runs directly on the hardware. From then on, a given physical processor is executing either the host world or the VMM world, with the VMDriver facilitating the transfer of control between the two worlds. A world switch between the VMM and the host worlds involves saving and restoring *all* user and system visible state on the CPU, and is thus more heavyweight than a normal process switch.

In this architecture, the CPU virtualization is handled by the VMM. A guest application or operating system performing pure computation runs just like a traditional mainframe-style virtual machine system. However, whenever the guest performs an I/O operation, the VMM will intercept it and switch to the host world rather than accessing the native hardware directly. Once in the host world, the VMAApp will perform the I/O on behalf of the virtual machine through appropriate system calls. For example, an attempt by the guest to fetch sectors from its disk will become a `read()` issued to the host for the corresponding data. The VMM also yields control to the host OS upon receiving a hardware interrupt. The hardware interrupt is reasserted in the host world so that the host OS will process the interrupt as if it came directly from hardware.

The hosted architecture is a powerful way for a PC-based virtual machine monitor to cope with the vast array of available hardware. One of the primary purposes of an

operating system is to present applications with an abstraction of the hardware that allows hardware-independent code to access the underlying devices. For example, a program to play audio CD-ROMs will work on both IDE and SCSI CD-ROM drives because operating systems provide an abstract CD-ROM interface. VMware Workstation takes advantage of this generality to run on whole classes of hardware without itself needing special device drivers for each possible device.

The most significant trade-off of a hosted architecture is in potential I/O performance degradation. Because I/O emulation is done in the host world, a virtual machine executing an I/O intensive workload can accrue extra CPU time switching between the VMM and host worlds, as well as significant time in the host world performing I/O to the native hardware. This increases the CPU overhead associated with any I/O operation.

Another trade-off of the hosted architecture is that the host OS is in full control of machine resources. Even though the VMM has full system and hardware privileges, it behaves cooperatively and allows the host OS to schedule it. The host OS can also page out the memory allocated to a particular virtual machine except for a small set of pages that the VMM has pinned on behalf of the virtual machine. This allows VMware Workstation to be treated by the host OS like a regular application, but occasionally at the expense of performance if the host OS makes poor resource scheduling choices for the virtual machine.

2.1 Virtualizing I/O Devices

Every VMware virtual machine is configured from the same set of potential virtual devices. Supported are standard PC devices such as a PS/2 keyboard, PS/2 mouse, floppy drive, IDE controllers with ATA disks and ATAPI CD-ROMs, a Soundblaster 16 sound card, and serial and parallel ports. Each virtual machine can also populate its virtual PCI slots with virtual BusLogic SCSI controllers, AMD PCNet™ Ethernet adapters, and an SVGA video controller for a special VMware virtual display card. Note that since the hardware besides the SVGA controller is made up of standard PC devices, existing guest operating system device drivers can communicate with it without modification.

In order to virtualize an I/O device, the VMM must be able to intercept all I/O operations issued by the guest operating system. On a PC, those accesses are generally done via special privileged IA-32 IN and OUT instructions. These are trapped by the VMM and emulated either in the VMM or the VMAApp by software that understands the semantics of the specific I/O port accessed. Any accesses that interact with the physical I/O hardware must be handled in the VMAApp, but the VMM can potentially handle accesses that do not interact with the hardware, *e.g.*,

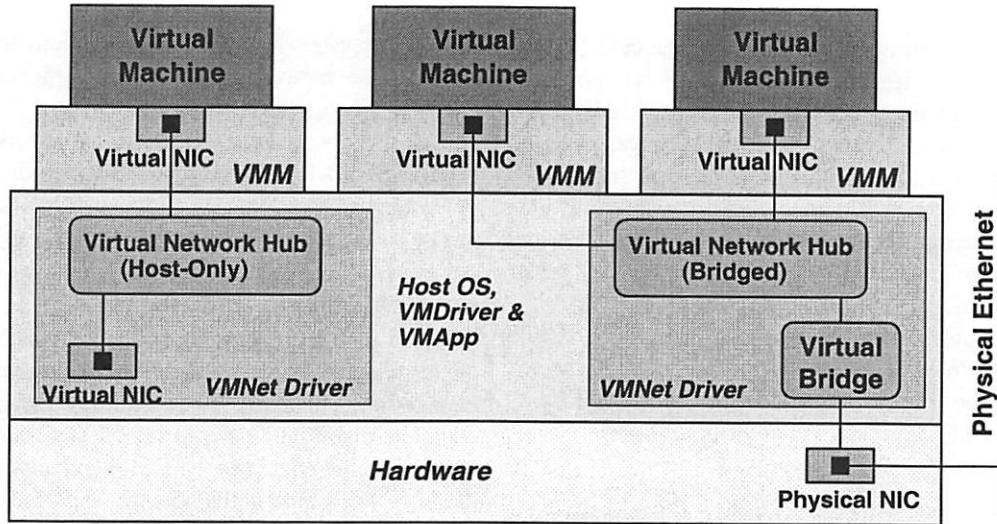


Figure 3: VMware’s network subsystem provides virtual Ethernet adapters, hubs and bridges. A hub can be either be bridged to a physical Ethernet adapter, or connected to a virtual network interface in the host OS. The virtual bridge and hub are implemented via a VMNet driver that is loaded into the host OS.

status ports or ports that merely latch data that will be used later. Restricting virtual devices to only a subset of available PC hardware greatly reduces the number of I/O ports that must be handled and the breadth of possibilities that handlers need to understand.

Virtualizing I/O devices with the hosted architecture can incur overhead from world switches between the VMM and the host, and even from the expense of handling the privileged instructions used to communicate with the hardware. However, these overheads matter only for devices with either high sustained throughput or low latency. The keyboard, for example, is perfectly suited to hosted virtualization.

2.2 Virtualizing a Network Card

An excellent example of a device that requires both high sustained throughput and low latency is a network interface card (NIC). Therefore, to understand how hosted device virtualization works and its performance implications, the following sections focus on the specific example of emulating a NIC in VMware Workstation. Figure 3 illustrates the components of the system. The virtual NIC appears to the guest as a full-fledged PCI Ethernet controller, complete with its own MAC address. The NIC emulation can be connected to the host in two ways— it can be bridged to the same physical network as a physical NIC or it can be connected to a virtual network created on the host. In both cases, the connection is implemented by a VMware VMNet driver that is loaded in the host operating system.

A virtual NIC that is bridged to a physical NIC is a true Ethernet bridge in the strictest sense. Its packets are sent on the wire with its own unique MAC address. The VMNet

driver runs the bridged physical NIC in promiscuous mode so that replies to that MAC address are picked up. The virtual NIC appears on the local Ethernet segment indistinguishably from any real machine. As a result, a virtual machine with a bridged virtual NIC can fully participate in accessing and providing network services.

A virtual NIC that is connected to a virtual network does not require an Ethernet interface on the host. Unlike the bridged case, the virtual network is completely private within the host and any participating virtual machines. If desired, the host OS can perform routing or IP masquerading to connect a virtual network to any type of external network, even to a non-Ethernet network. This paper will focus only on virtual NICs bridged to a physical NIC.

A virtual NIC itself is implemented via a combination of code in the VMM and the VMApp. The VMM exports a number of virtual I/O ports and a virtual IRQ that represent the virtual network adapter in the virtual machine. Reads and writes to these I/O ports, as well as virtual DMA transfers between the adapter and the virtual machine’s memory are semantically equivalent to those of a real network adapter. In VMware Workstation, the virtual NIC models an AMD Lance Am79C970A [1] controller, except that it is not limited to any specific network speed.

2.3 Sending and Receiving via a Virtualized NIC

Figure 4 depicts the components involved when sending and receiving packets via the hosted virtual NIC emulation described above. The guest operating system runs the device driver for a Lance controller. The driver initiates packet transmissions by reading and writing a sequence of virtual I/O ports, each of which switches back

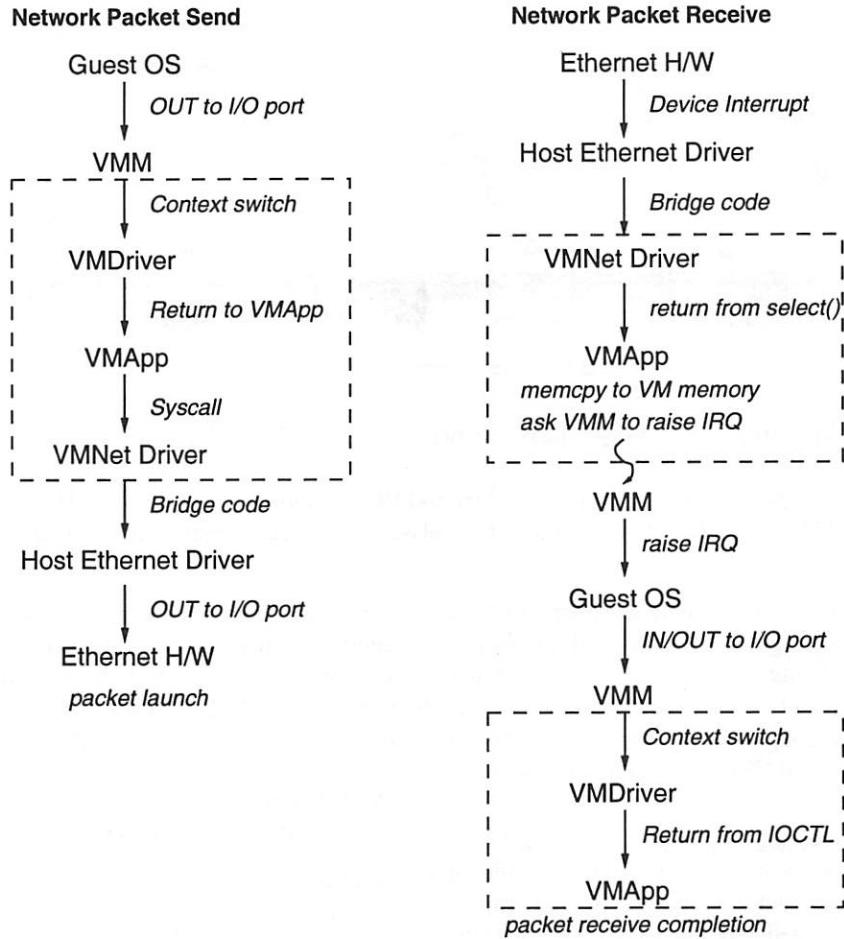


Figure 4: Components involved in a virtual machine network packet send and receive. Boxed components delineate components that are due to the hosted nature of the network device virtualization.

to the VMApp to emulate the Lance port accesses. On the final OUT instruction of the sequence, the Lance emulation does a normal `write()` to the VMNet driver, which passes the packet onto the network via a host NIC and then the VMApp switches back to the VMM, which raises a virtual IRQ to notify the guest device driver the packet was sent.

Packet receives occur in reverse. The bridged host NIC delivers the packet to the VMNet. The VMApp periodically runs `select()` on its connection to the VMNet and `read()`s the packet and requests that the VMM raise a virtual IRQ when it discovers any incoming packets. The VMM posts the virtual IRQ and the guest's Lance driver issues a sequence of I/O accesses to acknowledge the receipt to the hardware.

The boxed regions of the figure indicate extra work introduced by virtualizing the port accesses that actually send and receive packets. There is additional work in handling the intermediate I/O accesses and the privileged instructions associated with handling a virtual IRQ. Of the intermediate accesses, the ones to the virtual Lance's ad-

dress register are handled completely within the VMM and all accesses to the data register switch back to handling code in the VMApp.

This extra overhead consumes CPU cycles and increases the load on the CPU. The next section studies the effect of this extra overhead on I/O performance as well as CPU utilization. It breaks down the overheads along the boxed paths and describes overall time usage in the VMM and VMApp during the course of network activity.

3 Virtual Machine Networking Performance

A hosted virtualization strategy for I/O devices offers excellent flexibility and portability but at a potential tradeoff in performance for high throughput devices. Due to its nature, the hosted architecture incurs the following overheads: i) a world switch from the VMM to the host is required whenever the virtual machine needs to access real hardware, ii) I/O interrupt handling potentially involves the VMM, host OS, and guest OS interrupt handlers, iii) a packet transmission by the guest OS involves two device

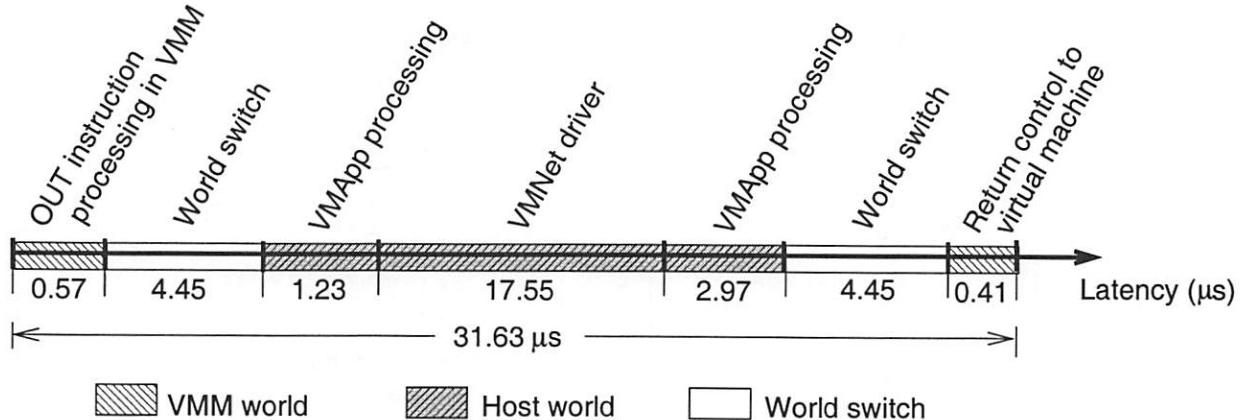


Figure 5: Microseconds spent along the path in the Host and VMM worlds processing an OUT instruction issued by the guest OS to a virtual AMD Lance NIC that initiates a physical network packet transmission on a 733 MHz CPU machine.

drivers - one in the guest and one on the host, and iv) there is an extra copy from the guest OS's physical memory to the host OS's kernel buffers on a packet transmit. Since these overheads consume CPU cycles, a system that is natively capable of saturating a high performance Ethernet link might instead become CPU bound when run within a virtual machine.

This section analyzes the overheads of sustained TCP transmits from a virtual machine. Experimental results of sustained TCP receives yield similar results and conclusions, and are not presented here due to space constraints. An analysis of these workloads exposes the major sources of virtualization overhead on a hosted architecture. Frequent switches between the host and VMM worlds is the most significant overhead. A set of optimizations targeted at these overheads improves the virtual networking subsystem substantially. The experimental results show that a set of three optimizations doubles sustained TCP transmit throughput on a slower machine that is CPU bound, and reduces CPU utilization significantly on a faster machine that is I/O bound.

3.1 Experimental Setup

The experiments were performed on two Intel-based PCs that are physically connected to each other via Intel EtherExpress 100 Mb/s Ethernet NICs and a direct, crossover cable:

PC-350 – a 350 MHz Pentium II system with 128 MBytes of RAM running a Linux 2.2.13 kernel

PC-733 – a 733 MHz Pentium III system with 256 MBytes of RAM running a Linux 2.2.17 kernel.

A virtual machine is configured with a virtual AMD Lance NIC bridged to the native Intel EtherExpress NIC.

The virtual machine runs a standard RedHat 6.2 Linux guest OS plus the 2.2.17-14 kernel update and uses the standard Linux pcnet32 driver to communicate over the virtual network. This virtual machine is hosted in two configurations on VMware Workstation 2.0:

VM/PC-350 – the virtual machine with 64 MBytes of RAM hosted by PC-350.

VM/PC-733 – the virtual machine with 128 MBytes of RAM hosted by PC-733.

The throughput experiments use a simple program called nettest that was developed internally for benchmarking network performance. The program opens a TCP connection between two IP addresses and copies a user specified amount of data with individual send()s and recv()s of a user specified size. The data transferred is merely repeated copies of the same in-memory buffer (to avoid paging and disk overhead) and is discarded on the receive side as soon as it arrives. The program measures the entire transfer and reports the average throughput in Mb/s.

3.2 Packet Transmit Overheads

The first series of experiments investigates the behavior of sustained virtual machine TCP transmits from **VM/PC-733** to **PC-350**. We configure nettest to send 100 megabytes using 4096-byte read()s and write()s. With VMware Workstation 2.0, we find that the workload is CPU bound with an average throughput over 30 consecutive runs of 64 Mb/s.

The workload is then instrumented to determine where the CPU time is spent. The first instrumentation gauges the time spent transmitting a packet by reading the Pentium Processor's *Time Stamp Counter* (TSC) register [10] at key points during the virtualization of the OUT instruction that

Total Time		
Category	Percent Time	Average Time
VMM Time	77.3%	N/A
Transmitting via the VMNet	8.7%	13.8 μ s
Emulating the Lance status register	4.0%	3.1 μ s
Handling host IRQs (device interrupts)	3.4%	N/A
Emulating the Lance transmit path	3.3%	5.2 μ s
Receiving via the VMNet	0.8%	1.8 μ s
VMM Time		
Category	Percent Time	Average Time
IN/OUTs requiring switching to the VMAApp	26.8%	7.45 μ s
Instructions not requiring virtualization	22.0%	N/A
General instructions requiring virtualization	11.6%	N/A
IN/OUTs handled in the VMM	8.3%	1.36 μ s
IN/OUTs to the Lance Address Port	8.1%	0.74 μ s
Transitioning to/from virtualization code	4.8%	N/A
Virtualizing the IRET instruction	4.8%	3.93 μ s
Delivering virtual IRQs (device interrupts)	4.6%	N/A

Table 1: Distribution of CPU time during network transmission. The largest overheads are I/O space accesses requiring a world switch to the VMAApp and the time spent handling them once in the VMAApp.

triggers a packet transmission. The TSC allows a measurement of the total cycle count of the path, plus internal breakdowns of interesting subsegments.

Figure 5 presents the latency involved along the instrumented network transmit path on **PC-733**. It takes a total of $0.57 + 4.45 + 1.23 + 17.55 = 23.8 \mu$ s from the start of the OUT instruction until the return from the VMNet system call that puts the packet on the wire. End-to-end, it takes 31.63μ s from the start of the OUT instruction that triggers a packet transmission until control is returned to the virtual machine and the next guest OS instruction is executed.

Of those 31.63μ s, 30.65μ s is spent in world switches and in the host world. Assuming the 17.55μ s of VMNet driver time in the host world is dominated by the unavoidable cost of actually transmitting the packet, we find that hosted virtualization architecture imposes $30.65 - 17.55 = 13.10 \mu$ s of overhead that would not be present if the VMM talked directly to the host NIC.

This overhead alone does not explain why the workload is CPU bound. At 31.63μ s per 1520-byte packet, it only takes roughly 0.26 seconds to transmit 100 megabits. Each packet transmission actually involves a series of 11 other IN/OUT instructions issued by the guest Ethernet driver as well as interrupt processing and virtualization overheads.

To investigate these other overheads, the next set of experiments uses time-based sampling to profile the distribution of time spent in the VMM and VMAApp over the entire workload. The samples measure the percentage of time spent in code sections and the number of samples that hit a section (when available). This gives a more com-

prehensive picture of the overheads present in transmitting packets and reveals some unnecessarily expensive paths. Table 1 summarizes the highest categories.

The profile shows that more than a quarter of the time in the VMM is spent preparing to call the VMAApp because of an I/O instruction, recording the result and then returning to the virtual machine. Additionally, each of those transitions also cost a world switch from the VMM to the host and back, which was calculated at around 8.90μ s on **PC-733** above (the switch time is part of the 77.3% running the VMM, but not part of any of the VMM Time numbers). Given that an I/O instruction on native hardware completes in a matter of tens of cycles, this is easily two orders of magnitude slower.

The other significant source of overhead is spread through the categories in Table 1: IRQ processing. The virtual AMD Lance NIC as well as the physical Intel EtherExpress NIC raises an IRQ (device interrupt) on every packet sent and received. Thus, the interrupt rate on the machine is very high for network-intensive workloads. On a hosted architecture, each IRQ that arrives while executing in the VMM world runs the VMM's interrupt handler then switches to the host world. The host world runs the host OS's interrupt handler for that IRQ, and passes control to the VMAApp to process any resulting actions. If the IRQ pertains to the guest (*e.g.*, the IRQ indicates that a packet was received that is destined for the guest), the VMAApp will then need to deliver a virtual IRQ to the guest OS. This involves switching back to the VMM world, delivering an IRQ to the virtual machine, and running the guest OS's interrupt handler.

This magnifies the cost of an IRQ since VMM and host interrupt handlers as well as guest interrupt handlers are run. Additionally, virtual interrupt handling routines execute privileged instructions that are expensive to virtualize. In Table 1, most of the IN/OUTs handled in the VMM are accesses to the virtual interrupt controller and the majority of the IRET instructions are the guest interrupt handler finishing. Note also that the cost of servicing an interrupt taken in the VMM world is much higher than servicing an interrupt taken in the host world due to the VMM interrupt handler and a world switch back to the host.

Yet another overhead in the hosted architecture which is not apparent from the raw profile is the inability of the VMAApp and VMM to distinguish between a hardware interrupt which produces an event for the virtual machine (*e.g.*, a packet to be delivered to the guest was received) from one that is unrelated to the virtual machine. Only the host OS and its drivers determine that. This leads to a balancing act: The VMAApp can do nothing when the VMM returns to the VMAApp on an IRQ, or it call `select()` in the VMAApp. Calling `select()` too frequently is wasteful, whereas calling `select()` too infrequently may cause harmful delays in handling network I/O events.

3.3 Reducing Network Virtualization Overheads

Guided by the results from the previous subsection that show world switch overheads as having the biggest impact, we implemented a set of optimizations aimed at reducing the number of world switches dramatically without departing from the hosted I/O architecture.

Handling I/O ports in the VMM Recall that the only virtual I/O accesses that require a world switch to the host are the ones that require a physical I/O device access. The vast majority of the I/O instructions are accesses to the Lance data port and only a third of them trigger packet transmissions. The remaining accesses merely modify the state of the virtual Lance data port, which can be easily done directly in the VMM without a world switch. Thus, an emulation of an OUT instruction that does not require real I/O can now be achieved in less than a tenth of the time it takes in VMware Workstation 2.0.

We also further reduce the cost of processing I/O accesses to the Lance address port by taking advantage of the property that the Lance address register has memory semantics, *i.e.*, reads and writes have no side effects and only latch the last value written. Thus, even though the instructions to access it are privileged instructions, the VMM can treat them as simple MOV instructions that happen to store to a special location. This allows the VMM to strip away several layers of virtualization and reduce the handling of the accesses to a handful of instructions.

Send combining The second optimization further reduces world switches by taking advantage of the fact that I/O intensive workloads have a high interrupt rate and the VMM must switch to the host whenever it takes a host IRQ. In VMware Workstation 2.0, each packet sent on the Lance adapter causes a world switch to the host to send the packet over the bridged network. Since part of the Lance data port emulation is now performed in the VMM, the VMM can delay the actual transmission until the next interrupt-induced switch to the host world.

Specifically, send combining work as follows: the VMM detects whether the system is experiencing a high world switch rate. If the rate (as recalculated periodically with an exponentially decaying counter) is high enough when the guest transmits a packet, the VMM queues it in a ring buffer and resumes the virtual machine. The next time a real interrupt occurs and control returns to the VMAApp it transmits any pending packets in the ring buffer. This effectively allows a packet transmission world switch to be combined with an interrupt-induced one.

Queueing the packets can be done without copying by leaving them in the virtual Lance controller's transmit ring buffer. If too many packets are delayed (currently defaulting to 3), the VMM will force a world switch to transmit the packets in order to insure that the native NIC is kept busy. In addition, there is a guaranteed world switch on the next IRQ from the host system timer so no packet will ever be delayed more than one tick (at which point the VMAApp will discontinue send combining if necessary). This optimization works well on I/O intensive workloads because interrupt rates are high enough that world switches are saved while I/O utilization is sustained.

Send combining also benefits both guest and host IRQs. Since the guest continues executing as soon as the packets are queued, there is a high probability that the guest will transmit multiple packets before the next mandatory world switch. This allows the VMAApp to process multiple transmit packets on a single world switch and deliver only a single virtual IRQ for the batch. As noted earlier, virtual IRQ delivery and the associated privileged virtualization are expensive operations. Furthermore, transmitting multiple packets at once increases the probability that native send-complete interrupts are taken while executing in the host world and hardware interrupts taken in the host world are serviced faster than those taken in the VMM world.

IRQ notification The third optimization is targeted at reducing host system calls for receiving notification of packet sends and receives. The VMAApp establishes a piece of shared memory with the VMNet driver at initialization and the driver sets a bitvector whenever packets are available. Then, on every NIC IRQ, instead of an expensive `select()` on all of the devices, the VMAApp checks the shared memory, receives any pending packets, and immediately returns to the VMM.

Total Time		
Category	Percent Time	Average Time
VMM Time	71.5%	N/A
Transmitting via the VMNet	17.9%	22.7 μ s
Receiving via the VMNet	2.5%	22.7 μ s
Emulating the Lance transmit path	1.8%	3.0 μ s
VMM Time		
Category	Percent Time	Average Time
Guest idle*	30.4%	N/A
Instructions not requiring virtualization	22.2%	N/A
Guest context switches	11.5%	N/A
Host IRQ processing while guest idle*	10.7%	N/A
Virtualizing privileged instructions	7.9%	N/A
IN/OUTs to the PIC (Interrupt Controller)	2.5%	0.78 μ s
Virtualization overheads of guest IRQs	2.5%	N/A
IN/OUTs to the Lance status register	2.3%	0.91 μ s
Transitioning to/from virtualization code	1.5%	N/A
IN/OUTs to the Lance that world switch	0.5%	N/A

Table 2: Distribution of CPU time during network transmission, with VMM optimizations. Many of the VMM time entries now represent a collection of individual instructions, which renders the Average Time not applicable. *Categories marked with “*” are partly derived from direct measurements presented in Section 3.5 for reasons described below.

In summary, the three major optimizations applied are as follows: Lance related I/O port accesses from the virtual machine are handled in the VMM whenever possible. During periods of heavy network activity, packet transmissions are merged and sent during IRQ-triggered world switches. This reduces the number of world switches, the number of virtual IRQs, and the number of host IRQs taken while executing in the VMM world. Finally, the VMNet driver is augmented with shared memory that allows the VMApp to avoid calling `select()` in some circumstances.

Figure 6 shows that these optimizations reduce CPU overhead enough to allow **VM/PC-733** to saturate a 100 Mbit Ethernet link, and the throughput for **VM/PC-350** more than doubles. Table 2 lists the CPU overhead breakdown from the time-based sampling measurements on **VM/PC-733** with the optimizations in place. Overall, the profile shows that the majority of the I/O related overhead is gone from the VMM and that there is now time when the guest OS idles. Additionally, guest context switch virtualization overheads now become significant as the guest switches between `nettest` and its idle task.

The “Guest idle” and “Host IRQ processing while guest idle” categories in Table 2 are derived with input from direct measurements presented in Section 3.5. A sample-based measurement of idle time indicates that 41.1% of VMM time is spent idling the guest and taking host IRQs while idling. However, discriminating the host IRQ processing time and guest idle time via time-based sampling alone is hard because of synchronized timer ticks and the heavy interrupt rate produced by the workload. We use di-

rect measurements that show that 21.7% of *total* time is spent in the guest idle loop to arrive at the idle time breakdown in Table 2.

The most effective optimization is handling IN and OUT accesses to Lance I/O ports directly in the VMM whenever possible. This eliminates world switches on Lance port access that do not require real I/O. Additionally, Table 1 indicates that accessing the Lance address register consumes around 8% of the VMM’s time and taking advantage of the register’s memory semantics has completely eliminated that overhead from the profile as shown in Table 2.

An interesting observation is that the time to transmit a packet via the VMNet does not change noticeably – all of the gains are along other paths. Instrumenting the optimized version in appropriate locations shows that the average cycle count on the path to transmit a packet onto the physical NIC is within 100 cycles of the totals from Figure 5. However, this is contrary to the times in Table 2 for sending via the VMNet driver. This disagreement stems from transmitting more than one packet at a time. While simply sending and timing individual packets, the baseline and optimized transmits look very similar, but with send combining active, up to 3 packets are sent back to back. This increases the chance of taking a host transmit IRQ from a prior transmit while in the VMNet driver. Since Table 2 reports the time from the start to finish of the call into the VMNet driver, it also includes the time the host kernel spends handling IRQs.

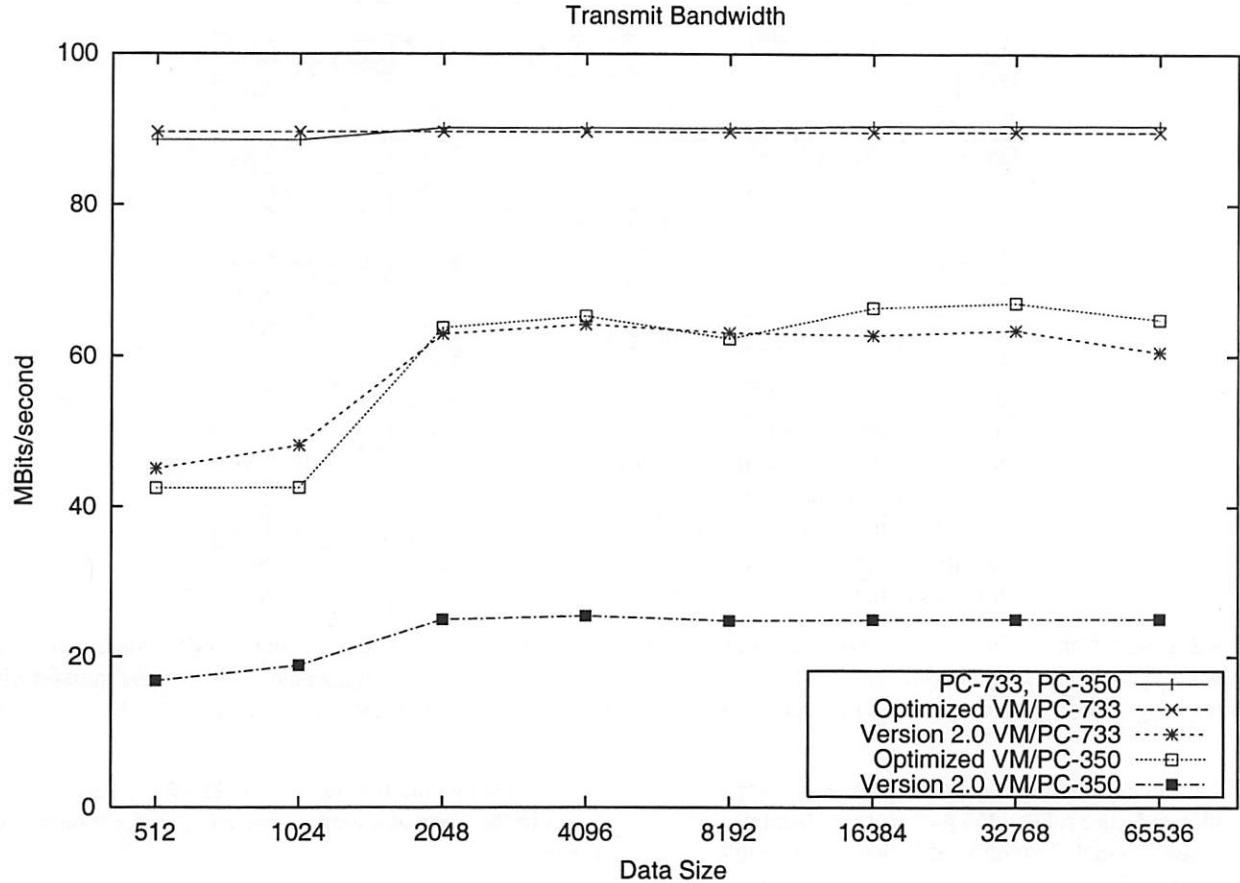


Figure 6: Throughput vs. Data Size when transmitting. The native and VM/PC-733 lines are transmitting to **PC-350** and the VM/PC-350 lines are transmitting to **PC-733**. The optimized VM/PC-733 was able to achieve native speeds, but non-I/O virtualization overheads limited VM/PC-350's achievable I/O throughput.

3.4 Throughput vs. Data Size: Transmit

The next series of experiments investigates the effect of the `per-write()` data size on the overall throughput. The data was gathered with the same `nettest` program and 100 MByte copies, but the amount of data copied per `write()` was varied from 512 to 65536 bytes by powers of two. 30 runs were taken and averaged at each data size with both the optimized and 2.0 versions. Figure 6 shows the sustained transmit throughput from the various machine configurations and optimization levels.

As expected, the native machines (both **PC-733** and **PC-350** achieve identical throughput) saturate the 100 Mbit link. **VM/PC-733** becomes CPU bound well before saturating the network link. With the optimizations however, **VM/PC-733** matches native throughput. Although **VM/PC-350** remains CPU bound with the optimizations, its sustained throughput doubles and matches the performance of the unoptimized **VM/PC-733**. The two **VM/PC-350** curves are consistent in shape with their **PC-733** counterparts.

3.5 CPU Utilization

Figure 6 shows that **VM/PC-733** is able to saturate a 100 Mbit link without becoming CPU bound, but **VM/PC-350** is CPU bound, even with optimizations. Natively, **PC-733** and **PC-350** easily saturate a 100 Mbit link. The final experiments set out to gather information about how utilized the CPU is in the different configurations.

We instrumented the system to obtain a precise measurement of idle time. Normally, when a guest issues a halt (`HLT`) instruction, VMware Workstation switches back to the VMAp which then blocks on a `select()` on all devices. Instead, we enabled an option whereby a guest `HLT` instruction spins and halts the CPU in the VMM rather than yielding control back to the host OS. Using the TSC register, we measure idle time starting from when the guest issues a `HLT` instruction to when the next hardware interrupt occurs. This idle time represents CPU cycles that is available to the guest OS for running other computation. Note that not all of this idle time would be available for other host OS computation, as there are a couple of world

Idle Time While Running nettest	
PC-733	86%
Optimized VM/PC-733	21.7%
Optimized VM/PC-733 without IRQ notification	17.9%
Optimized VM/PC-733 without send combining and IRQ notification	2.0%
Version 2.0 VM/PC-733	0%

Table 3: Percentage of total time spent idle for various configurations transmitting data on PC-733.

switches and some system call overhead (e.g., the `select()` system call) if we switched back to the VMApp on a guest HLT instruction.

For the native idle times, the standard profiler built into Linux kernels was augmented to account for time spent executing user code and in the kernel idle loop, and then the percentage of total ticks spent in the idle loop was taken.

The idle times in Table 3 show that in VM/PC-733, with a transmit size of 4KB, the guest has transitioned from being CPU bound at 64 Mb/s to being I/O bound with 21.7% idle time. In comparison, PC-733 has 86% idle time. At this point, nearly all of the remaining overheads are either part of CPU virtualization or part of the nature of the hosted architecture. The next section discusses further optimizations both within and outside the scope of a hosted architecture.

4 Performance Enhancements

The previous section showed that targeted optimizations can reduce the CPU overhead due to virtualization to the point where performance becomes I/O bound. This section describes strategies for further improving I/O performance and decreasing CPU utilization. The major areas for optimization include i) reducing CPU and interrupt controller virtualization overheads, ii) modifying the guest OS and/or its drivers, iii) modifying the host OS, and iv) accessing the native hardware directly from the virtual machine monitor. The last two techniques are departures from a pure hosted virtual machine architecture. Recall that the hosted architecture is designed with the requirement that existing host operating systems continue to run as usual, and that the virtualization software uses the host OS's API to access hardware devices.

4.1 Reducing CPU Virtualization Overhead

The optimized profile of Table 2 still shows significant overhead for “core CPU virtualization” overheads such as delivering virtual IRQs to a guest operating system, handling IRET instructions, and the MMU overheads associated with context switches. However, a discussion of any of these topics in enough detail to make concrete suggestions requires an understanding of VMware Workstation’s core virtualization technology, and is beyond the scope of this paper.

The profile does however suggest one easy optimization to reduce virtualization overhead. Guest OS accesses to the virtual PIC (interrupt controller) accounts for 2.5% of VMM time. A network card saturating a 100 megabit link is transmitting around 8000 packets per second and, in the case of TCP, receiving a steady flow of incoming ACK packets as well. This causes the virtual machine to receive a high rate of virtual IRQs. For each IRQ, the Linux guest IRQ handler issues five accesses to the virtual PIC. Since the virtual PIC is independent of the real PIC, it is handled in the VMM without requiring world switches. We can further optimize these accesses. One of the five accesses has memory semantics and can be inlined as a MOV instruction (just like the Lance address register). The other four accesses cause the current virtual PIC implementation to completely recalculate its internal state in a very general way – this can be specialized to reduce the overhead of those accesses.

4.2 Modifying the Guest OS

It is possible to modify the guest OS to avoid using instructions that virtualize inefficiently. Going a step further, it is also possible to provide a safe call into the VMM from the guest OS to provide some semantic knowledge about the guest to the VMM, or to perform some operations on its behalf. This technique comes at the price of compatibility with off-the-shelf guest OSes.

An optimization we tried in this category is to alter the Linux kernel to avoid page table switches when switching to the idle task. An idle guest spends a significant amount of time context switching to and from its idle task. A guest context switch operation uses a number of privileged instructions and changes guest page tables. This requires VMM intervention to implement the guest context switch safely. In the experiments above, as optimizations are added to reduce CPU utilization, the virtual machine execution profiles show an increasing fraction of CPU overhead due to virtualizing guest context switches. The VMM in VM/PC-733 spends 8.5% of its time virtualizing page tables switches.

Linux’s 2.2 kernels run the idle task as a kernel thread with the kernel’s page table. The kernel’s page table is a subset of every user application’s page table. This implies that it is not necessary to switch page tables when switch-

ing to the idle thread. Further, if the idle thread runs with the page table of the last user process to run and the idle thread ends up yielding back to the same process, another page table switch can be avoided. This optimization relies on trusting the idle thread not to corrupt user memory, a reasonable requirement since the idle thread runs at a trusted kernel-level.

We prototyped the optimization of running the idle task with the prior user application’s page table by modifying the Linux kernel’s context switch function. This modification halves the MMU derived virtualization overhead, and almost all of the saved CPU cycles become CPU idle time. Besides reducing virtualization overhead, such an optimization may also benefit software-based IA-32 CPU implementations where the overhead of emulating the instructions involved in a context switch is significant.

4.3 Optimizing the Guest Driver Protocol

A hosted architecture allows the NIC emulation code to communicate to the host via an abstracted interface that is independent of the host’s native hardware. It is possible to design a similarly abstracted imaginary Ethernet controller whose interface is an idealization designed explicitly to virtualize well. For example, the Linux pcnet32 driver issues 12 I/O instructions and takes one IRQ for every single packet transmitted. An idealized virtual NIC could use only a single OUT to indicate a packet is ready to send and completely skip the transmit IRQ, or to only get an IRQ when space becomes available in the array of outgoing packets. The idealized device can also arrange its transmit and receive buffers very simply in memory rather than with the elaborate flexibility, but complexity of the Lance controller’s buffers. In fact, VMware’s server products support a vmxnet network adapter that implements such an ideal interface.

The major drawback of creating an idealized virtual NIC is the need for custom device drivers for every guest OS. Since the AMD Lance is a well supported NIC, most operating systems already include drivers for it. These existing drivers work *unmodified* in the guest OSes. Any idealized NIC would need to have an array of its own drivers written, distributed, and maintained. Thus, while an idealized driver is a potential accelerating option, it is likely feasible only for critical situations on a select group of guest operating systems, such as in a server environment.

4.4 Modifying the Host OS

Just as expensive virtualization overheads can sometimes be removed by modifying the guest rather than by modifying the VMM, some bumps in the hosted architecture’s handling of networking are best smoothed by modifying the host. One promising change is to expand the ways in which the Linux networking stack allocates and handles

`sk_buffs`. Each time the VMApp sends a packet via the VMNet driver, the driver allocates an `sk_buff` and copies the data from the VMApp into the `sk_buff`. The Linux kernel profiler shows that a very significant portion of the time spent in the host kernel while running the network transmit workload is due to copying data from the VMApp into an `sk_buff`.

In Linux, `sk_buff` creation uses `kmalloc()` to allocate the data area. If a driver could specify its own data region, then it would be possible to transmit packets via the VMNet driver without the copy. The driver would need to be responsible for making sure that its allocated `sk_buffs` are neither leaked nor freed too early. However, for the VMnet driver, the backing for the `sk_buff` data area would come from the memory representing the virtual machine’s physical memory. This memory would be at least as persistent as the virtual machine itself, and any packets transmitted via a VMNet would presumably only be interesting as long as their corresponding virtual machine exists.

The primary disadvantage of modifying the host OS is that it requires the cooperation of OS vendors, or, in the case of Linux, the active support of Linux kernel maintainers. Otherwise the optimization will not be available on unmodified off-the-shelf host OSes.

4.5 Bypassing the Host OS

As long as actual transmits to and receives from the physical network require a world switch back to the host operating system and the VMApp, an unavoidable latency and CPU overhead will remain. Additionally, the VMM will have to take native IRQs while running, world switch them back to the host, and wait for incoming packets to work through the host and VMApp before they reach the guest. *This fundamentally limits the I/O performance of a hosted virtual machine architecture.* To truly maximize I/O bandwidth, the VMM must drive the I/O device directly. The guest OS could potentially drive the device directly too, but this requires either hardware support or memory access restrictions to preserve safety and isolation.

With its own device drivers, the VMM can send and receive packets without any mandatory world switches and relay receive IRQs to the guest almost immediately. Additionally, there would be no need for a separate VMNet driver. However, adding device drivers to the VMM represents a major trade-off. Recall that VMware Workstation supports a wide variety of hardware devices because of the hosted architecture. It automatically gains support for new I/O devices and bug-fixes in existing drivers as soon as the host OS does. A VMM that requires its own NIC drivers would require an investment of resources in developing, testing, and updating its hardware support.

As described, each VMM is associated with a single

virtual machine. In order to share an I/O device among several virtual machines, the VMM would have to be extended to include a global component that recognizes the individual virtual machines and their VMMs. The global component would effectively be a kernel that is specifically designed for managing VMM worlds. In addition to driving the device, the global component would have to provide software to multiplex more than one VMM onto a single I/O device. This technique is used in VMware ESX Server™, where achieving native I/O performance for high-speed devices is an important requirement.

5 Related Work

Providing interoperability and preserving compatibility are frequently necessary when introducing any new technology. As computer architectures and operating systems advance, they need to remain compatible with existing software and applications. By providing a hardware abstraction layer, virtual machine technology allows hardware differences to be hidden from legacy software, and allows multiple incompatible computing environments to co-exist on a machine.

Achieving native machine performance is a prime target of virtual machine technology. The ability to execute virtual machine code directly on the hardware allows the technology to outperform other technologies based on machine simulation or emulation. Subsequent to the early mainframe virtual machine support, IBM designed a number of architectural features to further enhance the performance of virtual machines. Gum [7] describes a number of hardware assists in the IBM System/370 architecture for further reducing the overhead of handling privileged guest instructions, guest memory address translation, and multi-processing support.

Borden *et al.* [2] describe PR/SM, a partitioning feature on the IBM 3090 series of mainframes that allows specific devices, I/O channels and memory address ranges to be dedicated to a virtual machine. Guest I/O accesses can then be handled directly by the hardware without requiring VMM intervention. Borden *et al.* report that this feature allows a virtual machine with dedicated I/O devices to achieve within 1–2% of native hardware performance. A PC-based server platform with similar partitioning features would allow VMware's virtual machines to do the same.

Hall and Robinson [8] describe virtualizing the VAX architecture which, like the IA-32 architecture, is not naturally virtualizable and has more than two protection rings. They rely on modifications to the VAX architecture as well as the microcode. In contrast, VMware's virtualization technology does not require any hardware modifications.

Bugnion *et al.* [3] apply virtual machine technology towards providing scalable performance on large scale NUMA machines. Most commodity operating systems

do not scale to a large NUMA machine without extensive modifications. However, a virtual machine monitor can be designed from the ground up to manage such a machine and hide its NUMA nature from a commodity OS. The machine can then run multiple commodity OS images, with each OS allocated as many CPUs as it can scale to.

VMware Workstation's hosted virtual machine architecture relies on user-level emulation of I/O devices. This parallels the approach taken by microkernel-based operating systems (*e.g.*, Mach [6]) which rely on user-level emulation of operating system APIs to provide multiple application environments on a single machine. The primary difference lies at the abstraction layer: while virtual machines abstract the hardware layer, microkernels abstract the OS API layer. Härtig *et al.* [9] describe techniques for improving the performance of microkernel-based systems.

6 Summary and Conclusions

This paper describes VMware's hosted virtual machine architecture as implemented in VMware Workstation. This architecture enables VMware Workstation to support a wide variety of PC hardware without special device drivers and to present a constant and hence portable virtual hardware environment. Additionally, co-existing with an commodity operating system simplifies installation and use for users and reduces the complexity of the virtual machine monitor component for the developers.

The hosted architecture splits its functionality between a VMM component that virtualizes the CPU, and a VMApp component that runs as a normal application on a host OS and handles I/O to the native devices on behalf of a virtual machine. I/O intensive workloads, in addition to running significant amounts of privileged code, require heavy-weight world switches from the VMM back to the VMApp on the host. While this is unimportant for low bandwidth devices like keyboards or mice, it can potentially prevent more demanding devices from achieving the same I/O saturation as their native counterparts. This paper focuses specifically on NIC virtualization. It presents optimizations to VMware Workstation 2.0 that allow a virtual machine hosted on a 733 MHz Pentium III CPU to saturate the network without becoming CPU bound.

The key strategy behind all the implemented optimizations is to reduce the number of world switches. The first optimization takes advantage of the fact that only a fraction of the I/O accesses to the virtual NIC causes packets to be transmitted. The remainder do not require any access to the host hardware, allowing the VMM to handle them directly instead of switching back to the host world. This optimization alone reduces CPU utilization to the point where the network link is completely saturated on a 733 MHz CPU.

The second optimization reduces the remaining world switches and trims their overhead. When the world switch

rate is high enough, rather than switch back to the VMApp immediately to send each packet, the VMM gathers up to 3 packets at a time before switching back to the VMApp to send them all at once. An extra benefit of this clustering is that transmit IRQs from the native NIC becomes more likely to arrive in the host world (while sending successive packets) than in the VMM world where they would require an immediate world switch.

The third optimization uses shared memory between the VMNet driver and the VMApp to reduce the need to issue `select()` calls from the VMApp. This optimization allows the VMApp to detect which NIC IRQ requires contacting the VMNet and which NIC IRQ can immediately switch back to the VMM without spending extra time in the VMApp. Together, these three optimizations reduce the CPU utilization of the 733 MHz CPU virtual machine to around 78%. The optimizations also more than double the achievable network throughput on a 350 MHz CPU virtual machine.

The experimental results confirm that CPU overheads of a hosted virtualization strategy can prevent an I/O intensive virtual machine workload from matching the performance of the same workload on native hardware. In the straightforward implementation, frequent I/O causes frequent world switches that artificially limit the I/O utilization because the workload becomes CPU bound. However, even while remaining within a hosted virtual machine architecture, we are able to eliminate spurious world switches and even restructure around seemingly mandatory crossings with significant reduction in CPU utilization to the point that a 733 MHz Pentium III system is I/O bound with plenty of CPU cycles to spare.

CPUs are constantly getting faster and a 733 MHz Pentium III is at or below entry level for today's corporate PCs. Further, very few desktop workloads saturate a full 100 Mbit link with any regularity or frequency. Taken in conjunction with the portability, device independence, and co-existence a hosted architecture provides, VMware Workstation's achievable I/O performance strikes a good balance between performance and compatibility for its target desktop usage. The balance may change of course when gigabit networks become prevalent, depending on how fast CPUs will be by then.

Acknowledgments

VMware Workstation's hosted virtual machine architecture is the brainchild of Mendel Rosenblum, Edouard Bugnion, Scott Devine and Edward Wang. Regis Duschenne provided several optimizations to the network subsystem. The anonymous referees provided useful feedback that improved the paper. Finally, this paper would not be possible without the contributions of the dedicated employees of VMware, Inc.

References

- [1] AMD Corporation, Sunnyvale, CA. *Network Products: Ethernet Controllers Book 2*, 1998.
- [2] Terry L. Borden, James P. Hennessy, and James W. Rymarczyk. Multiple operating systems on one processor complex. *IBM Systems Journal*, 28(1):104–123, 1989.
- [3] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commercial Operating Systems on Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.
- [4] R. J. Creasy. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, 25(5):483–490, September 1981.
- [5] Robert. P. Goldberg. Survey of virtual machine research. *IEEE Computer Magazine*, 7(6):34–45, 1974.
- [6] David Golub, Randall Dean, Allesandro Forin, and Richard Rashid. Unix as an application program. In *Proceedings of the USENIX 1990 Summer Conference*, June 1990.
- [7] Peter H. Gum. System/370 Extended Architecture: Facilities for Virtual Machines. *IBM Journal of Research and Development*, 27(6):530–544, November 1983.
- [8] Judith S. Hall and Paul T. Robinson. Virtualizing the VAX Architecture. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 380–389. ACM, May 1991.
- [9] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The Performance of μ -Kernel-Based Systems. In *Proceedings of the Sixteenth Symposium on Operating System Principles*. ACM, October 1997.
- [10] Intel Corporation, Santa Clara, CA. *Intel Architecture Developer's Manual. Volumes I, II and III*, 1998.
- [11] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources

Jeff Bonwick, *Sun Microsystems*

Jonathan Adams, *California Institute of Technology*

Abstract

The slab allocator [Bonwick94] provides efficient object caching but has two significant limitations: its global locking doesn't scale to many CPUs, and the allocator can't manage resources other than kernel memory. To provide scalability we introduce a per-processor caching scheme called the *magazine layer* that provides linear scaling to any number of CPUs. To support more general resource allocation we introduce a new virtual memory allocator, *vmem*, which acts as a universal backing store for the slab allocator. *Vmem* is a complete general-purpose resource allocator in its own right, providing several important new services; it also appears to be the first resource allocator that can satisfy arbitrary-size allocations in constant time. Magazines and *vmem* have yielded performance gains exceeding 50% on system-level benchmarks like LADDIS and SPECweb99.

We ported these technologies from kernel to user context and found that the resulting *libumem* outperforms the current best-of-breed user-level memory allocators. *libumem* also provides a richer programming model and can be used to manage other user-level resources.

1. Introduction

The slab allocator [Bonwick94] has taken on a life of its own since its introduction in these pages seven years ago. Initially deployed in Solaris 2.4, it has since been adopted in whole or in part by several other operating systems including Linux, FreeBSD, NetBSD, OpenBSD, EROS, and Nemesis. It has also been adapted to applications such as BIRD and Perl. Slab allocation is now described in several OS textbooks [Bovet00, Mauro00, Vahalia96] and is part of the curriculum at major universities worldwide.

Meanwhile, the Solaris slab allocator has continued to evolve. It now provides per-CPU memory allocation, more general resource allocation, and is available as a user-level library. We describe these developments in seven sections as follows:

§2. Slab Allocator Review. We begin with brief review of the original slab allocator.

§3. Magazines: Per-CPU Memory Allocation. As servers with many CPUs became more common and memory latencies continued to grow relative to processor speed, the slab allocator's original locking strategy became a performance bottleneck. We addressed this by introducing a per-CPU caching scheme called the *magazine layer*.

§4. Vmem: Fast, General Resource Allocation. The slab allocator caches relatively small objects and relies on a more general-purpose backing store to provide slabs and satisfy large allocations. We describe a new resource allocator, *vmem*, that can manage arbitrary sets of integers – anything from virtual memory addresses to minor device numbers to process IDs. *Vmem* acts as a universal backing store for the slab allocator, and provides powerful new interfaces to address more complex resource allocation problems. *Vmem* appears to be the first resource allocator that can satisfy allocations and frees of any size in guaranteed constant time.

§5. Vmem-Related Slab Allocator Improvements. We describe two key improvements to the slab allocator itself: it now provides object caching for *any* *vmem* arena, and can issue *reclaim callbacks* to notify clients when the arena's resources are running low.

§6. libumem: A User-Level Slab Allocator. We describe what was necessary to transplant the slab allocator from kernel to user context, and show that the resulting *libumem* outperforms even the current best-of-breed multithreaded user-level allocators.

§7. Conclusions. We conclude with some observations about how these technologies have influenced Solaris development in general.

2. Slab Allocator Review

2.1. Object Caches

Programs often cache their frequently used objects to improve performance. If a program frequently allocates and frees `foo` structures, it is likely to employ highly optimized `foo_alloc()` and `foo_free()` routines to “avoid the overhead of malloc.” The usual strategy is to cache `foo` objects on a simple freelist so that most allocations and frees take just a handful of instructions. Further optimization is possible if `foo` objects naturally return to a partially initialized state before they’re freed, in which case `foo_alloc()` can assume that an object on the freelist is already partially initialized.

We refer to the techniques described above as *object caching*. Traditional malloc implementations cannot provide object caching because the malloc/free interface is typeless, so the slab allocator introduced an explicit object cache programming model with interfaces to create and destroy object caches, and allocate and free objects from them (see Figure 2.1).

The allocator and its clients cooperate to maintain an object’s partially initialized, or *constructed*, state. The allocator guarantees that an object will be in this state when allocated; the client guarantees that it will be in this state when freed. Thus, we can allocate and free an object many times without destroying and reinitializing its locks, condition variables, reference counts, and other invariant state each time.

2.2. Slabs

A *slab* is one or more pages of virtually contiguous memory, carved up into equal-size chunks, with a reference count indicating how many of those chunks are currently allocated. To create new objects the allocator creates a slab, applies the *constructor* to each chunk, and adds the resulting objects to the cache. If system memory runs low the allocator can reclaim any slabs whose reference count is zero by applying the *destructor* to each object and returning memory to the VM system. Once a cache is populated, allocations and frees are very fast: they just move an object to or from a freelist and update its slab reference count.

Figure 2.1: Slab Allocator Interface Summary

```
kmem_cache_t *kmem_cache_create(
    char *name,                      /* descriptive name for this cache */
    size_t size,                      /* size of the objects it manages */
    size_t align,                     /* minimum object alignment */
    int (*constructor)(void *obj, void *private, int kmflag),
    void (*destructor)(void *obj, void *private),
    void (*reclaim)(void *private),   /* memory reclaim callback */
    void *private,                   /* argument to the above callbacks */
    vmem_t *vmp,                     /* vmem source for slab creation */
    int cflags);                    /* cache creation flags */
```

Creates a cache of objects, each of size `size`, aligned on an `align` boundary. `name` identifies the cache for statistics and debugging. `constructor` and `destructor` convert plain memory into objects and back again; `constructor` may fail if it needs to allocate memory but can’t. `reclaim` is a callback issued by the allocator when system-wide resources are running low (see §5.2). `private` is a parameter passed to the `constructor`, `destructor` and `reclaim` callbacks to support parameterized caches (e.g. a separate packet cache for each instance of a SCSI HBA driver). `vmp` is the *vmem source* that provides memory to create slabs (see §4 and §5.1). `cflags` indicates special cache properties. `kmem_cache_create()` returns an opaque pointer to the object cache (a.k.a. *kmem cache*).

```
void kmem_cache_destroy(kmem_cache_t *cp);
```

Destroys the cache and releases all associated resources. All allocated objects must have been freed.

```
void *kmem_cache_alloc(kmem_cache_t *cp, int kmflag);
```

Gets an object from the cache. The object will be in its constructed state. `kmflag` is either `KM_SLEEP` or `KM_NOSLEEP`, indicating whether it’s acceptable to wait for memory if none is currently available.

```
void kmem_cache_free(kmem_cache_t *cp, void *obj);
```

Returns an object to the cache. The object must be in its constructed state.

3. Magazines

"Adding per-CPU caches to the slab algorithm would provide an excellent allocator."

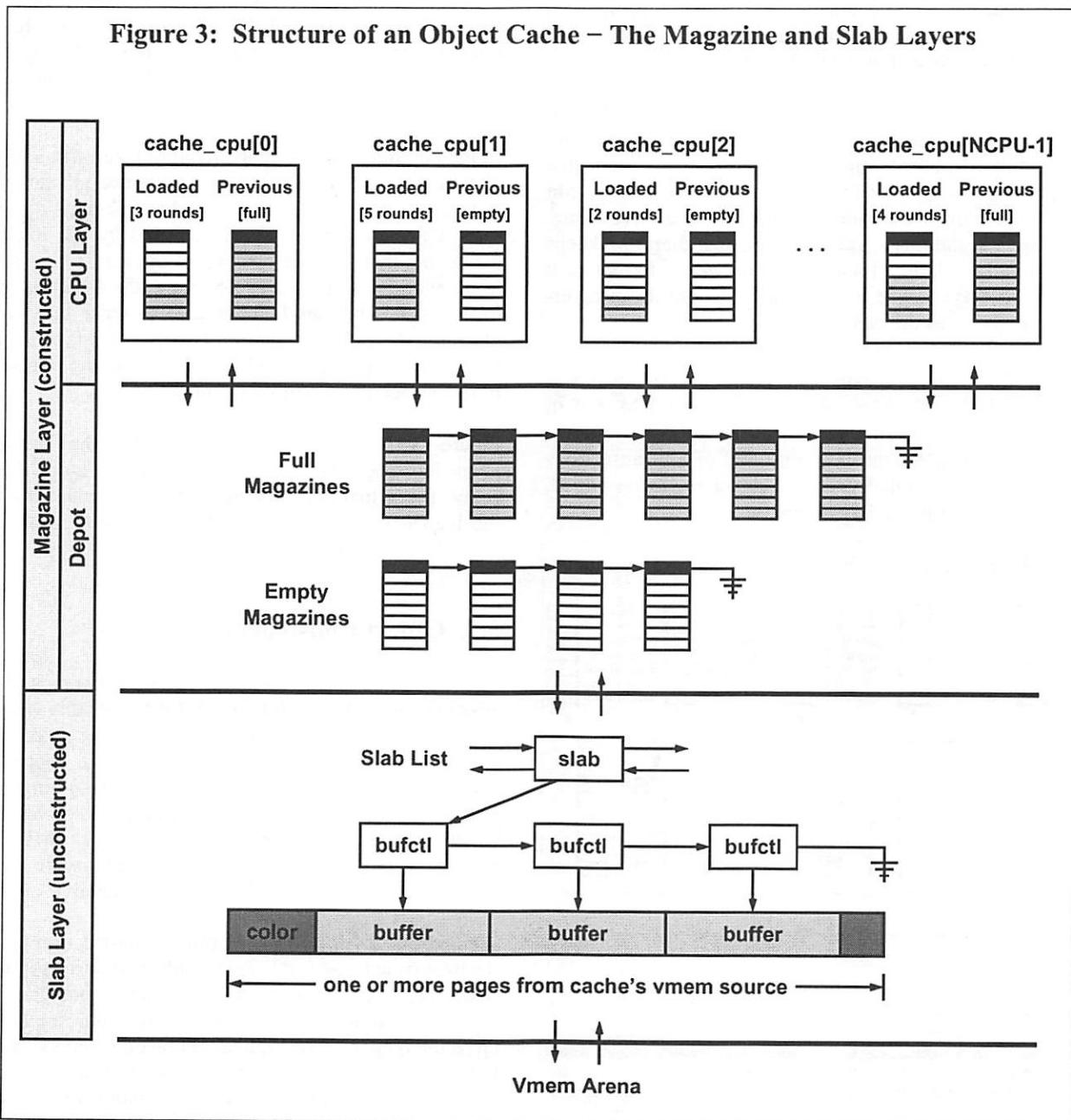
Uresh Vahalia, *UNIX Internals: The New Frontiers*

The biggest limitation of the original slab allocator is that it lacks multiprocessor scalability. To allocate an object the allocator must acquire the lock that protects the cache's slab list, thus serializing all allocations. To allow all CPUs to allocate in parallel we need some form of per-CPU caching.

Our basic approach is to give each CPU an M -element cache of objects called a *magazine*, by analogy with automatic weapons. Each CPU's magazine can satisfy M allocations before the CPU needs to *reload* – that is, exchange its empty magazine for a full one. The CPU doesn't access any global data when allocating from its magazine, so we can increase scalability arbitrarily by increasing the magazine size (M).

In this section we describe how the magazine layer works and how it performs in practice. Figure 3 (below) illustrates the key concepts.

Figure 3: Structure of an Object Cache – The Magazine and Slab Layers



3.1. Overview

A *magazine* is an M -element array of pointers to objects* with a count of the number of *rounds* (valid pointers) currently in the array. Conceptually, a magazine works like a stack. To allocate an object from a magazine we pop its top element:

```
obj = magazine[--rounds];
```

To free an object to a magazine we push it on top:

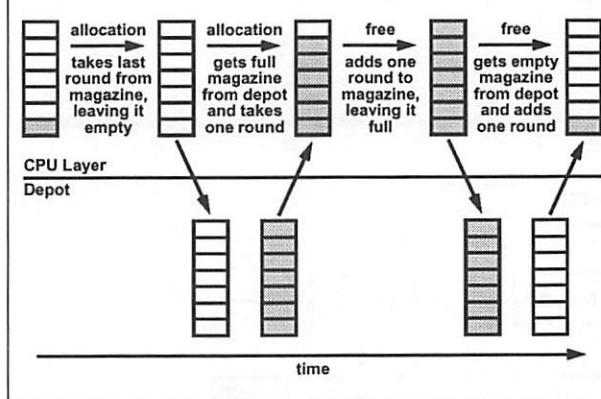
```
magazine[rounds++] = obj;
```

We use magazines to provide each object cache with a small per-CPU object supply. Each CPU has its own *loaded magazine*, so transactions (allocations and frees) can proceed in parallel on all CPUs.

The interesting question is what to do if the loaded magazine is empty when we want to allocate an object (or full when we want to free one). We cannot just fall through to the slab layer, because then a long run of allocations would miss in the CPU layer every time, ruining scalability. Each object cache therefore keeps a global stockpile of magazines, the *depot*, to replenish its CPU layer. We refer to the CPU and depot layers collectively as the *magazine layer*.

With M -round magazines we would intuitively expect the CPU layer's miss rate to be at most $1/M$, but in fact a tight loop of two allocations followed by two frees can cause thrashing, with half of all transactions accessing the globally-locked depot *regardless of M*, as shown in Figure 3.1a below.

Figure 3.1a: Thrashing at a Magazine Boundary



*We use an array of object pointers, rather than just linking objects together on a freelist, for two reasons: first, freelist linkage would overwrite an object's constructed state; and second, we plan to use the slab allocator to manage arbitrary resources, so we can't assume that the objects we're managing are backed by writable memory.

We address this by keeping the *previously loaded magazine* in the CPU layer, as shown in Figure 3 (previous page). If the loaded magazine cannot satisfy a transaction but the previous magazine can, we exchange *loaded* with *previous* and try again. If neither magazine can satisfy the transaction, we return *previous* to the depot, move *loaded* to *previous*, and load a new magazine from the depot.

The key observation is that the only reason to load a new magazine is to replace a full with an empty or vice versa, so we know that after each reload the CPU either has a full *loaded magazine* and an empty *previous magazine* or vice versa. The CPU can therefore satisfy at least M allocations and at least M frees entirely with CPU-local magazines before it must access the depot again, so the CPU layer's worst-case miss rate is bounded by $1/M$ regardless of workload.

In the common case of short-lived objects with a high allocation rate there are two performance advantages to this scheme. First, balanced alloc/free pairs on the same CPU can almost all be satisfied by the loaded magazine; therefore we can expect the actual miss rate to be even lower than $1/M$. Second, the LIFO nature of magazines implies that we tend to reuse the same objects over and over again. This is advantageous in hardware because the CPU will already own the cache lines for recently modified memory.

Figure 3.1b (next page) summarizes the overall magazine algorithm in pseudo-code. Figure 3.1c shows the actual code for the hot path (i.e. hitting in the loaded magazine) to illustrate how little work is required.

3.2. Object Construction

The original slab allocator applied constructors at slab creation time. This can be wasteful for objects whose constructors allocate additional memory. To take an extreme example, suppose an 8-byte object's constructor attaches a 1K buffer to it. Assuming 8K pages, one slab would contain about 1000 objects, which after construction would consume 1MB of memory. If only a few of these objects were ever allocated, most of that 1MB would be wasted.

We addressed this by moving object construction up to the magazine layer and keeping only raw buffers in the slab layer. Now a buffer becomes an object (has its constructor applied) when it moves from the slab layer up to the magazine layer, and an object becomes a raw buffer (has its destructor applied) when it moves from the magazine layer back down to the slab layer.

Figure 3.1b: The Magazine Algorithm

The allocation and free paths through the magazine layer are almost completely symmetric, as shown below. The only asymmetry is that the free path is responsible for populating the depot with empty magazines, as explained in §3.3.

Alloc: <pre> if (the CPU's loaded magazine isn't empty) pop the top object and return it; if (the CPU's previous magazine is full) exchange loaded with previous, goto Alloc; if (the depot has any full magazines) return previous to depot, move loaded to previous, load the full magazine, goto Alloc; allocate an object from the slab layer, apply its constructor, and return it; </pre>	Free: <pre> if (the CPU's loaded magazine isn't full) push the object on top and return; if (the CPU's previous magazine is empty) exchange loaded with previous, goto Free; if (the depot has any empty magazines) return previous to depot, move loaded to previous, load the empty magazine, goto Free; if (an empty magazine can be allocated) put it in the depot and goto Free; apply the object's destructor and return it to the slab layer </pre>
---	---

Figure 3.1c: The Hot Path in the Magazine Layer

```

void *
kmem_cache_alloc(kmem_cache_t *cp, int kmflag)
{
    kmem_cpu_cache_t *ccp = &cp->cache_cpu[CPU->cpu_id];

    mutex_enter(&ccp->cc_lock);
    if (ccp->cc_rounds > 0) {
        kmem_magazine_t *mp = ccp->cc_loaded;
        void *obj = mp->mag_round[--ccp->cc_rounds];
        mutex_exit(&ccp->cc_lock);
        return (obj);
    }
    ...
}

void
kmem_cache_free(kmem_cache_t *cp, void *obj)
{
    kmem_cpu_cache_t *ccp = &cp->cache_cpu[CPU->cpu_id];

    mutex_enter(&ccp->cc_lock);
    if (ccp->cc_rounds < ccp->cc_magsize) {
        kmem_magazine_t *mp = ccp->cc_loaded;
        mp->mag_round[ccp->cc_rounds++] = obj;
        mutex_exit(&ccp->cc_lock);
        return;
    }
    ...
}

```

3.3. Populating the Magazine Layer

We have described how the magazine layer works once it's populated, but how does it *get* populated?

There are two distinct problems here: we must allocate objects, and we must allocate magazines to hold them.

- **Object allocation.** In the allocation path, if the depot has no full magazines, we allocate a single object from the slab layer and construct it.
- **Magazine allocation.** In the free path, if the depot has no empty magazines, we allocate one.

We never allocate full magazines explicitly, because it's not necessary: empty magazines are eventually filled by frees, so it suffices to create empty magazines and let full ones emerge as a side effect of normal allocation/free traffic.

We allocate the magazines themselves (i.e. the arrays of pointers) from object caches, just like everything else; there is no need for a special magazine allocator.*

3.4. Dynamic Magazine Resizing

Thus far we have discussed M-element magazines without specifying how M is determined. We've observed that we can make the CPU layer's miss rate as low as we like by increasing M, but making M larger than necessary would waste memory. We therefore seek the smallest value of M that delivers linear scalability.

Rather than picking some "magic value," we designed the magazine layer to tune itself dynamically. We start each object cache with a small value of M and observe the contention rate on the depot lock. We do this by using a non-blocking trylock primitive on the depot lock; if that fails we use the ordinary blocking lock primitive and increment a contention count. If the contention rate exceeds a fixed threshold we increase the cache's magazine size. We enforce a maximum magazine size to ensure that this feedback loop can't get out of control, but in practice the algorithm behaves extremely well on everything from desktops to 64-CPU Starfires. The algorithm generally stabilizes after several minutes of load with reasonable magazine sizes and depot lock contention rates of less than once per second.

*Note that if we allocated full magazines in the allocation path, this would cause infinite recursion the first time we tried to allocate a magazine for one of the magazine caches. There is no such problem with allocating empty magazines in the free path.

3.5. Protecting Per-CPU State

An object cache's CPU layer contains per-CPU state that must be protected either by per-CPU locking or by disabling interrupts. We selected per-CPU locking for several reasons:

- **Programming Model.** Some operations, such as changing a cache's magazine size, require the allocator to modify the state of each CPU. This is trivial if the CPU layer is protected by locks.
- **Real-time.** Disabling interrupts increases dispatch latency (because it disables preemption), which is unacceptable in a real-time operating system like Solaris [Khanna92].
- **Performance.** On most modern processors, grabbing an uncontended lock is cheaper than modifying the processor interrupt level.

3.6. Hardware Cache Effects

Even per-CPU algorithms don't scale if they suffer from *false sharing* (contention for ownership of a cache line that can occur when multiple CPUs modify logically unrelated data that happens to reside in the same physical cache line). We carefully pad and align the magazine layer's per-CPU data structures so that each one has its own cache line. We found that doing so is *critical* for linear scalability on modern hardware.

An allocator can also *induce* false sharing by handing out objects smaller than a cache line to more than one CPU [Berger00]. We haven't found this to be a problem in practice, however, because most kernel data structures are larger than a cache line.

3.7. Using the Depot as a Working Set

When the system is in steady state, allocations and frees must be roughly in balance (because memory usage is roughly constant). The variation in memory consumption over a fixed period of time defines a form of working set [Denning68]; specifically, it defines how many magazines the depot must have on hand to keep the allocator working mostly out of its high-performance magazine layer. For example, if the depot's full magazine list varies between 37 and 47 magazines over a given period, then the working set is 10 magazines; the other 37 are eligible for reclaiming.

The depot continuously tracks the working set sizes of its full and empty magazine lists, but does not actually free excess magazines unless memory runs low.

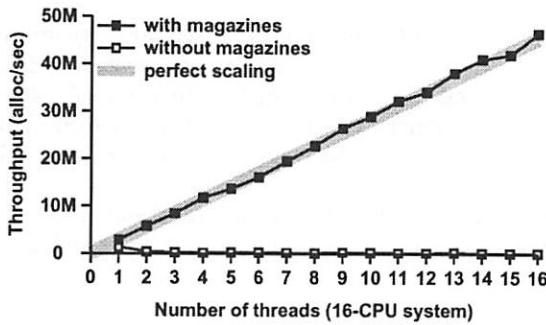
3.8. Microbenchmark Performance

The two key metrics for an MT-hot memory allocator are latency and scalability. We measured latency as the average time per iteration of a tight alloc/free loop. We measured scalability by running multiple instances of the latency test on a 333MHz 16-CPU Starfire.

The latency test revealed that the magazine layer improves even single-CPU performance (356ns per alloc/free pair vs. 743ns for the original slab allocator) because the hot path is so simple (see Figure 3.1c). Indeed, there is little room for further improvement in latency because the cost of locking imposes a lower bound of 186ns.

As we increased the number of threads the magazine layer exhibited perfect linear scaling, as shown below. Without the magazine layer, throughput was actually *lower* with additional threads due to increasingly pathological lock contention. With 16 threads (all 16 CPUs busy) the magazine layer delivered 16 times higher throughput than a single CPU (and 340 times higher throughput than the original allocator), with the same 356ns latency.

Figure 3.8: Allocation Scalability



3.9. System-Level Performance

We ran several system-level benchmarks both with and without the magazine layer to assess the magazine layer's effectiveness.* The system was uniformly faster with magazines, with the greatest improvements in allocator-intensive workloads like network I/O.

*Unfortunately we could not make direct comparisons with other kernel memory allocators because the Solaris kernel makes extensive use of the object cache interfaces, which are simply not available in other allocators. We will, however, provide direct comparisons with best-of-breed user-level allocators in §6.

3.9.1. SPECweb99

We ran the industry-standard SPECweb99 web server benchmark [SPEC01] on an 8-CPU E4500. The magazine layer more than *doubled* performance, from 995 to 2037 simultaneous connections. The gain is so dramatic because every network packet comes from the allocator.

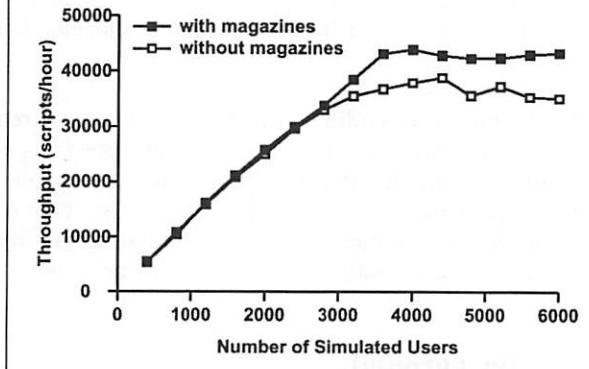
3.9.2. TPC-C

We ran the industry-standard TPC-C database benchmark [TPC01] on an 8-CPU E6000. Magazines improved performance by 7%. The gain here is much more modest than with SPECweb99 because TPC-C is not very demanding of the kernel memory allocator.

3.9.3. Kenbus

We ran Kenbus, a precursor to the SPEC SMT (System Multi-Tasking) benchmark currently under development [SPEC01], on a 24-CPU E6000. The magazine layer improved peak throughput by 13% and improved the system's ability to *sustain* peak throughput as load increased. At maximum tested load (6000 users) the magazine layer improved system throughput by 23%.

Figure 3.9.3: Kenbus Performance



3.10. Summary

The magazine layer provides efficient object caching with very low latency and linear scaling to any number of CPUs. We discussed the magazine layer in the context of the slab allocator, but in fact the algorithms are completely general. A magazine layer can be added to *any* memory allocator to make it scale.

4. Vmem

The slab allocator relies on two lower-level system services to create slabs: a virtual address allocator to provide kernel virtual addresses, and VM routines to back those addresses with physical pages and establish virtual-to-physical translations.

Incredibly, we found that our largest systems were scalability-limited by the old virtual address allocator. It tended to fragment the address space badly over time, its latency was linear in the number of fragments, and the whole thing was single-threaded.

Virtual address allocation is just one example of the more general problem of *resource allocation*. For our purposes, a *resource* is anything that can be described by a set of integers. For example, virtual addresses are subsets of the 64-bit integers; process IDs are subsets of the integers [0, 30000]; and minor device numbers are subsets of the 32-bit integers.

Resource allocation (in the sense described above) is a fundamental problem that every operating system must solve, yet it is surprisingly absent in the literature. It appears that 40 years of research on memory allocators has simply never been applied to resource allocators. The resource allocators for Linux, all the BSD kernels, and Solaris 7 or earlier all use linear-time algorithms.

In this section we describe a new general-purpose resource allocator, *vmem*, which provides guaranteed constant-time performance with low fragmentation. *Vmem* appears to be the first resource allocator that can do this.

We begin by providing background on the current state of the art. We then lay out our objectives in creating *vmem*, describe the *vmem* interfaces, explain the implementation in detail, and discuss *vmem*'s performance (fragmentation, latency, and scalability) under both benchmarks and real-world conditions.

4.1. Background

Almost all versions of Unix have a *resource map allocator* called `rmalloc()` [Vahalia96]. A resource map can be any set of integers, though it's most often an address range like [0xe0000000, 0xf0000000]. The interface is simple: `rmalloc(map, size)` allocates a segment of the specified `size` from `map`, and `rmfree(map, size, addr)` gives it back.

Linux's *resource allocator* and BSD's *extent allocator* provide roughly the same services. All three suffer from serious flaws in both design and implementation:

- **Linear-time performance.** All three allocators maintain a list of free segments, sorted in address order so the allocator can detect when *coalescing* is possible: if segments [a, b) and [b, c) are both free, they can be merged into a single free segment [a, c) to reduce fragmentation. The allocation code performs a linear search to find a segment large enough to satisfy the allocation. The free code uses insertion sort (also a linear algorithm) to return a segment to the free segment list. It can take several *milliseconds* to allocate or free a segment once the resource becomes fragmented.

- **Implementation exposure.** A resource allocator needs data structures to keep information about its free segments. In various ways, all three allocators make this *your problem*:

- `rmalloc()` requires the creator of the resource map to specify the maximum possible number of free segments at map creation time. If the map ever gets more fragmented than that, the allocator throws away resources in `rmfree()` because it has nowhere to put them. (!)
- Linux puts the burden on its *clients* to supply a segment structure with each allocation to hold the allocator's internal data. (!)
- BSD allocates segment structures dynamically, but in so doing creates an awkward failure mode: `extent_free()` fails if it can't allocate a segment structure. It's difficult to deal with an allocator that won't let you give stuff back.

We concluded that it was time to abandon our stone tools and bring modern technology to the problem.

4.2. Objectives

We believe a good resource allocator should have the following properties:

- A powerful interface that can cleanly express the most common resource allocation problems.
- Constant-time performance, regardless of the size of the request or the degree of fragmentation.
- Linear scalability to any number of CPUs.
- Low fragmentation, even if the operating system runs at full throttle for *years*.

We'll begin by discussing the interface considerations, then drill down to the implementation details.

4.3. Interface Description

The vmem interfaces do three basic things: create and destroy *arenas* to describe resources, allocate and free resources, and allow arenas to *import* new resources dynamically. This section describes the key concepts and the rationale behind them. Figure 4.3 (next page) provides the complete vmem interface specification.

4.3.1. Creating Arenas

The first thing we need is the ability to define a resource collection, or *arena*. An arena is simply a set of integers. Vmem arenas most often represent virtual memory addresses (hence the name *vmem*), but in fact they can represent any integer resource, from virtual addresses to minor device numbers to process IDs.

The integers in an arena can usually be described as a single contiguous range, or *span*, such as [100, 500], so we specify this *initial span* to `vmem_create()`. For discontiguous resources we can use `vmem_add()` to piece the arena together one span at a time.

- **Example.** To create an arena to represent the integers in the range [100, 500] we can say:

```
foo = vmem_create("foo", 100, 400, ...);
```

(Note: 100 is the start, 400 is the size.) If we want `foo` to represent the integers [600, 800] as well, we can add the span [600, 800] by using `vmem_add()`:

```
vmem_add(foo, 600, 200, VM_SLEEP);
```

`vmem_create()` specifies the arena's natural unit of currency, or *quantum*, which is typically either 1 (for single integers like process IDs) or `PAGESIZE` (for virtual addresses). Vmem rounds all sizes to quantum multiples and guarantees quantum-aligned allocations.

4.3.2. Allocating and Freeing Resources

The primary interfaces to allocate and free resources are simple: `vmem_alloc(vmp, size, vmflag)` allocates a *segment* of *size* bytes from arena `vmp`, and `vmem_free(vmp, addr, size)` gives it back.

We also provide a `vmem_xalloc()` interface that can specify common *allocation constraints*: *alignment*, *phase* (offset from the alignment), *address range*, and *boundary-crossing restrictions* (e.g. “don't cross a page boundary”). `vmem_xalloc()` is useful for things like kernel DMA code, which allocates kernel virtual addresses using the phase and alignment constraints to ensure correct cache coloring.

- **Example.** To allocate a 20-byte segment whose address is 8 bytes away from a 64-byte boundary, and which lies in the range [200, 300], we can say:

```
addr = vmem_xalloc(foo, 20, 64, 8, 0,  
                    200, 300, VM_SLEEP);
```

In this example `addr` will be 262: it is 8 bytes away from a 64-byte boundary ($262 \bmod 64 = 8$), and the segment [262, 282] lies within [200, 300].

Each `vmem_[x]alloc()` can specify one of three *allocation policies* through its `vmflag` argument:

- **VM_BESTFIT.** Directs vmem to use the smallest free segment that can satisfy the allocation. This policy tends to minimize fragmentation of very small, precious resources.
- **VM_INSTANTFIT.** Directs vmem to provide a good approximation to best-fit in guaranteed constant time. This is the default allocation policy.
- **VM_NEXTFIT.** Directs vmem to use the next free segment after the one previously allocated. This is useful for things like process IDs, where we want to cycle through all the IDs before reusing them.

We also offer an arena-wide allocation policy called *quantum caching*. The idea is that most allocations are for just a few quanta (e.g. one or two pages of heap or one minor device number), so we employ high-performance caching for each multiple of the quantum up to `qcache_max`, specified in `vmem_create()`. We make the caching threshold explicit so that each arena can request the amount of caching appropriate for the resource it manages. Quantum caches provide *perfect-fit*, very low latency, and linear scalability for the most common allocation sizes (§4.4.4).

4.3.3. Importing From Another Arena

Vmem allows one arena to *import* its resources from another. `vmem_create()` specifies the *source arena*, and the functions to allocate and free from that source. The arena imports new spans as needed, and gives them back when all their segments have been freed.

The power of importing lies in the *side effects* of the import functions, and is best understood by example. In Solaris, the function `segkmem_alloc()` invokes `vmem_alloc()` to get a virtual address and then backs it with physical pages. Therefore, we can create an arena of mapped pages by simply importing from an arena of virtual addresses using `segkmem_alloc()` and `segkmem_free()`. Appendix A illustrates how vmem's import mechanism can be used to create complex resources from simple building blocks.

Figure 4.3: Vmem Interface Summary

```

vmem_t *vmem_create(
    char *name,                                /* descriptive name */
    void *base,                                 /* start of initial span */
    size_t size,                               /* size of initial span */
    size_t quantum,                            /* unit of currency */
    void *(*afunc)(vmem_t *, size_t, int),     /* import alloc function */
    void (*ffunc)(vmem_t *, void *, size_t),   /* import free function */
    vmem_t *source,                            /* import source arena */
    size_t qcache_max,                         /* maximum size to cache */
    int vmflag);                             /* VM_SLEEP or VM_NOSLEEP */

Creates a vmem arena called name whose initial span is [base, base + size). The arena's natural
unit of currency is quantum, so vmem_alloc() guarantees quantum-aligned results. The arena may
import new spans by invoking afunc on source, and may return those spans by invoking ffunc on
source. Small allocations are common, so the arena provides high-performance caching for each
integer multiple of quantum up to qcache_max. vmflag is either VM_SLEEP or VM_NOSLEEP
depending on whether the caller is willing to wait for memory to create the arena. vmem_create()
returns an opaque pointer to the arena.

void vmem_destroy(vmem_t *vmp);

Destroys arena vmp.

void *vmem_alloc(vmem_t *vmp, size_t size, int vmflag);

Allocates size bytes from vmp. Returns the allocated address on success, NULL on failure.
vmem_alloc() fails only if vmflag specifies VM_NOSLEEP and no resources are currently available.
vmflag may also specify an allocation policy (VM_BESTFIT, VM_INSTANTFIT, or VM_NEXTFIT) as
described in §4.3.2. If no policy is specified the default is VM_INSTANTFIT, which provides a good
approximation to best-fit in guaranteed constant time.

void vmem_free(vmem_t *vmp, void *addr, size_t size);

Frees size bytes at addr to arena vmp.

void *vmem_xalloc(vmem_t *vmp, size_t size, size_t align, size_t phase,
size_t nocross, void *minaddr, void *maxaddr, int vmflag);

Allocates size bytes at offset phase from an align boundary such that the resulting segment
[addr, addr + size) is a subset of [minaddr, maxaddr) that does not straddle a nocross-
aligned boundary. vmflag is as above. One performance caveat: if either minaddr or maxaddr is
non-NULL, vmem may not be able to satisfy the allocation in constant time. If allocations within a
given [minaddr, maxaddr) range are common it is more efficient to declare that range to be its own
arena and use unconstrained allocations on the new arena.

void vmem_xfree(vmem_t *vmp, void *addr, size_t size);

Frees size bytes at addr, where addr was a constrained allocation. vmem_xfree() must be used if
the original allocation was a vmem_xalloc() because both routines bypass the quantum caches.

void *vmem_add(vmem_t *vmp, void *addr, size_t size, int vmflag);

Adds the span [addr, addr + size) to arena vmp. Returns addr on success, NULL on failure.
vmem_add() will fail only if vmflag is VM_NOSLEEP and no resources are currently available.

```

4.4. Vmem Implementation

In this section we describe how vmem actually works. Figure 4.4 illustrates the overall structure of an arena.

4.4.1. Keeping Track of Segments

"Apparently, too few researchers realized the full significance of Knuth's invention of boundary tags."

Paul R. Wilson et. al. in [Wilson95]

Most implementations of `malloc()` prepend a small amount of space to each buffer to hold information for the allocator. These *boundary tags*, invented by Knuth in 1962 [Knuth73], solve two major problems:

- They make it easy for `free()` to determine how large the buffer is, because `malloc()` can store the size in the boundary tag.
- They make coalescing trivial. Boundary tags link all segments together in address order, so `free()` can simply look both ways and coalesce if either neighbor is free.

Unfortunately, resource allocators can't use traditional boundary tags because the resource they're managing may not be memory (and therefore may not be able to hold information). In vmem we address this by using

external boundary tags. For each segment in the arena we allocate a boundary tag to manage it, as shown in Figure 4.4 below. (See Appendix A for a description of how we allocate the boundary tags themselves.) We'll see shortly that external boundary tags enable constant-time performance.

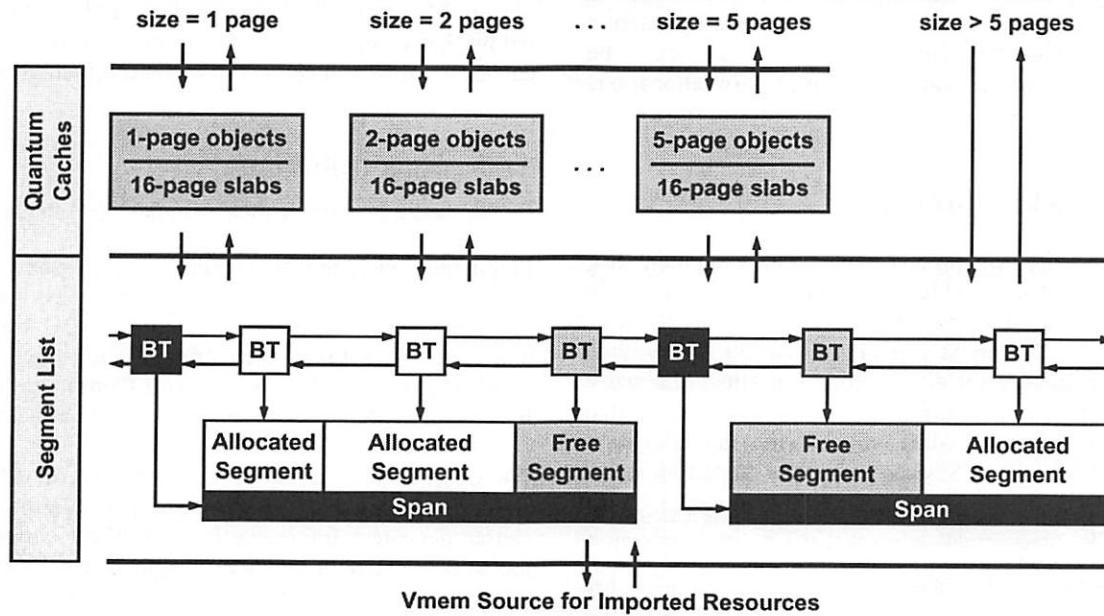
4.4.2. Allocating and Freeing Segments

Each arena has a *segment list* that links all of its segments in address order, as shown in Figure 4.4. Every segment also belongs to either a freelist or an allocation hash chain, as described below. (The arena's segment list also includes *span markers* to keep track of span boundaries, so we can easily tell when an imported span can be returned to its source.)

We keep all free segments on power-of-two freelists; that is, `freelist[n]` contains all free segments whose sizes are in the range $[2^n, 2^{n+1})$. To allocate a segment we search the appropriate freelist for a segment large enough to satisfy the allocation. This approach, called *segregated fit*, actually approximates *best-fit* because *any* segment on the chosen freelist is a *good fit* [Wilson95]. (Indeed, with power-of-two freelists, a segregated fit is necessarily within 2x of a *perfect fit*.) Approximations to *best-fit* are appealing because they exhibit low fragmentation in practice for a wide variety of workloads [Johnstone97].

Figure 4.4: Structure of a Vmem Arena

`vmem_alloc()` vectors allocations based on size: small allocations go to the quantum caches, larger ones to the segment list. In this figure we've depicted an arena with a 1-page quantum and a 5-page `qcache_max`. Note that the "segment list" is, strictly speaking, a list of boundary tags ("BT" below) that *represent* the segments. Boundary tags for allocated segments (white) are also linked into an allocated-segment hash table, and boundary tags for free segments (gray) are linked into size-segregated freelists (not shown).



The algorithm for selecting a free segment depends on the allocation policy specified in the flags to `vmem_alloc()` as follows; in all cases, assume that the allocation size lies in the range $[2^n, 2^{n+1})$:

- **VM_BESTFIT.** Search for the smallest segment on `freelist[n]` that can satisfy the allocation.
- **VM_INSTANTFIT.** If the size is exactly 2^n , take the first segment on `freelist[n]`. Otherwise, take the first segment on `freelist[n+1]`. Any segment on this freelist is necessarily large enough to satisfy the allocation, so we get constant-time performance with a reasonably good fit.*
- **VM_NEXTFIT.** Ignore the freelists altogether and search the arena for the next free segment after the one previously allocated.

Once we've selected a segment, we remove it from its freelist. If the segment is not an exact fit we split the segment, create a boundary tag for the remainder, and put the remainder on the appropriate freelist. We then add our newly-allocated segment's boundary tag to a hash table so `vmem_free()` can find it quickly.

`vmem_free()` is straightforward: it looks up the segment's boundary tag in the allocated-segment hash table, removes it from the hash table, tries to coalesce the segment with its neighbors, and puts it on the appropriate freelist. All operations are constant-time. Note that the hash lookup also provides a cheap and effective sanity check: the freed address must be in the hash table, and the freed size must match the segment size. This helps to catch bugs such as duplicate frees.

The key feature of the algorithm described above is that its performance is independent of both transaction size *and* arena fragmentation. Vmem appears to be the first resource allocator that can perform allocations and frees of any size in guaranteed constant time.

4.4.3. Locking Strategy

For simplicity, we protect each arena's segment list, freelists, and hash table with a global lock. We rely on the fact that large allocations are relatively rare, and allow the arena's quantum caches to provide linear scalability for all the common allocation sizes. This strategy is very effective in practice, as illustrated by the performance data in §4.5 and the allocation statistics for a large Solaris 8 server in Appendix B.

*We like instant-fit because it guarantees constant time performance, provides low fragmentation in practice, and is easy to implement. There are many other techniques for choosing a suitable free segment in reasonable (e.g. logarithmic) time, such as keeping all free segments in a size-sorted tree; see [Wilson95] for a thorough survey. Any of these techniques could be used for a vmem implementation.

4.4.4. Quantum Caching

The slab allocator can provide object caching for any vmem arena (§5.1), so vmem's quantum caches are actually implemented as object caches. For each small integer multiple of the arena's quantum we create an object cache to service requests of that size. `vmem_alloc()` and `vmem_free()` simply convert each small request ($\text{size} \leq \text{qcache_max}$) into a `kmem_cache_alloc()` or `kmem_cache_free()` on the appropriate cache, as illustrated in Figure 4.4. Because it is based on object caching, quantum caching provides very low latency and linear scalability for the most common allocation sizes.

- **Example.** Assume the arena shown in Figure 4.4. A 3-page allocation would proceed as follows: `vmem_alloc(foo, 3 * PAGESIZE)` would call `kmem_cache_alloc(foo->vm_qcache[2])`. In most cases the cache's magazine layer would satisfy the allocation, and we would be done. If the cache needed to create a new slab it would call `vmem_alloc(foo, 16 * PAGESIZE)`, which would be satisfied from the arena's segment list. The slab allocator would then divide its 16-page slab into five 3-page objects and use one of them to satisfy the original allocation.

When we create an arena's quantum caches we pass a flag to `kmem_cache_create()`, KMC_QCACHE, that directs the slab allocator to use a particular slab size: the next power of two above $3 * \text{qcache_max}$. We use this particular value for three reasons: (1) the slab size *must* be larger than `qcache_max` to prevent infinite recursion; (2) by numerical luck, this slab size provides near-perfect slab packing (e.g. five 3-page objects fill 15/16 of a 16-page slab); and (3) we'll see below that using a common slab size for all quantum caches helps to reduce overall arena fragmentation.

4.4.5. Fragmentation

"A waste is a terrible thing to mind." — Anonymous

Fragmentation is the disintegration of a resource into unusably small, discontiguous segments. To see how this can happen, imagine allocating a 1GB resource one byte at a time, then freeing only the even-numbered bytes. The arena would then have 500MB free, yet it could not even satisfy a 2-byte allocation.

We observe that it is the *combination* of different allocation sizes and different segment lifetimes that causes persistent fragmentation. If all allocations are the same size, then any freed segment can obviously satisfy another allocation of the same size. If all allocations are transient, the fragmentation is transient.

We have no control over segment lifetime, but quantum caching offers some control over allocation size: namely, all quantum caches have the same slab size, so most allocations from the arena's segment list occur in slab-size chunks.

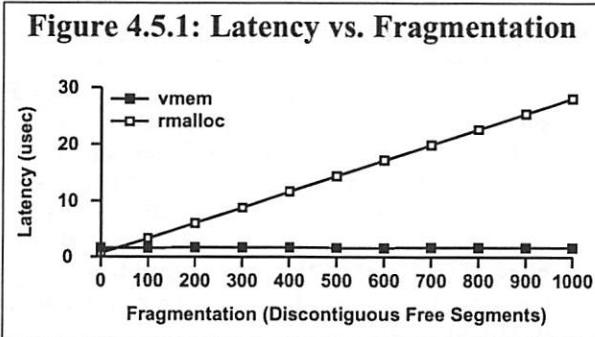
At first it may appear that all we've done is move the problem: the segment list won't fragment as much, but now the quantum caches *themselves* can suffer fragmentation in the form of partially-used slabs. The critical difference is that the free objects in a quantum cache are *of a size that's known to be useful*, whereas the segment list can disintegrate into *useless* pieces under hostile workloads. Moreover, prior allocation is a good predictor of future allocation [Weinstock88], so free objects are likely to be used again.

It is impossible to *prove* that this helps,* but it seems to work well in practice. We have never had a report of severe fragmentation since vmem's introduction (we had many such reports with the old resource map allocator), and Solaris systems often stay up for *years*.

4.5. Performance

4.5.1. Microbenchmark Performance

We've stated that `vmem_alloc()` and `vmem_free()` are constant-time operations regardless of arena fragmentation, whereas `rmalloc()` and `rmfree()` are linear-time. We measured alloc/free latency as a function of fragmentation to verify this.



`rmalloc()` has a slight performance edge at very low fragmentation because the algorithm is so naïve. At zero fragmentation, vmem's latency *without quantum caching* was 1560ns, vs. 715ns for `rmalloc()`. Quantum caching reduces vmem's latency to just 482ns, so for allocations that go to the quantum caches (the common case) vmem is faster than `rmalloc()` even at very low fragmentation.

*In fact, it has been proven that "there is no reliable algorithm for ensuring efficient memory usage, and none is possible." [Wilson95]

4.5.2. System-Level Performance

Vmem's low latency and linear scaling remedied serious pathologies in the performance of kernel virtual address allocation under `rmalloc()`, yielding dramatic improvements in system-level performance.

- **LADDIS.** Veritas reported a 50% improvement in LADDIS peak throughput with the new virtual memory allocator [Taylor99].
- **Web Service.** On a large Starfire system running 2700 Netscape servers under Softway's Share II scheduler, vmem reduced system time from 60% to 10%, roughly doubling system capacity [Swain98].
- **I/O Bandwidth.** An internal I/O benchmark on a 64-CPU Starfire generated such heavy contention on the old `rmalloc()` lock that the system was essentially useless. Contention was exacerbated by very long hold times due to `rmalloc()`'s linear search of the increasingly fragmented kernel heap. `lockstat(1M)` (a Solaris utility that measures kernel lock contention) revealed that threads were spinning for an average of 48 *milliseconds* to acquire the `rmalloc()` lock, thus limiting I/O bandwidth to just $1000/48 = 21$ I/O operations per second per CPU. With vmem the problem completely disappeared and performance improved by several *orders of magnitude*.

4.6. Summary

The vmem interface supports both simple and highly constrained allocations, and its *importing* mechanism can build complex resources from simple components. The interface is sufficiently general that we've been able to eliminate over 30 special-purpose allocators in Solaris since vmem's introduction.

The vmem implementation has proven to be very fast and scalable, improving performance on system-level benchmarks by 50% or more. It has also proven to be very robust against fragmentation in practice.

Vmem's *instant-fit policy* and *external boundary tags* appear to be new concepts. They guarantee constant-time performance regardless of allocation size or arena fragmentation.

Vmem's *quantum caches* provide very low latency and linear scalability for the most common allocations. They also present a particularly friendly workload to the arena's segment list, which helps to reduce overall arena fragmentation.

5. Core Slab Allocator Enhancements

Sections 3 and 4 described the magazine and vmem layers, two new technologies above and below the slab layer. In this section we describe two vmem-related enhancements to the slab allocator itself.

5.1. Object Caching for Any Resource

The original slab allocator used `rmalloc()` to get kernel heap addresses for its slabs and invoked the VM system to back those addresses with physical pages.

Every object cache now uses a vmem arena as its slab supplier. The slab allocator simply invokes `vmem_alloc()` and `vmem_free()` to create and destroy slabs. It makes no assumptions about the nature of the resource it's managing, so it can provide object caching for *any* arena.* This feature is what makes vmem's high-performance *quantum caching* possible (§4.4.4).

5.2. Reclaim Callbacks

For performance, the kernel caches things that aren't strictly needed. The DNLC (directory name lookup cache) improves pathname resolution performance, for example, but most DNLC entries aren't actually in use at any given moment. If the DNLC could be notified when the system was running low on memory, it could free some of its entries to relieve memory pressure.

We support this by allowing clients to specify a *reclaim callback* to `kmem_cache_create()`. The allocator calls this function when the cache's vmem arena is running low on resources. The callback is purely advisory; what it actually does is entirely up to the client. A typical action might be to give back some fraction of the objects, or to free all objects that haven't been accessed in the last N seconds.

This capability allows clients like the DNLC, inode cache and NFS_READDIR cache to grow more or less unrestricted until the system runs low on memory, at which point they are asked to start giving some back.

One possible future enhancement would be to add an argument to the reclaim callback to indicate the number of bytes wanted, or the "level of desperation." We have not yet done so because simple callback policies like "give back 10% each time I'm called" have proven to be perfectly adequate in practice.

*For caches backed by non-memory vmem arenas, the caller must specify the KMC_NOTOUCH flag to `kmem_cache_create()` so the allocator won't try to use free buffers to hold its internal state.

6. User-Level Memory Allocation: The libumem Library

It was relatively straightforward to transplant the magazine, slab, and vmem technologies to user-level. We created a library, *libumem*, that provides all the same services. In this section we discuss the handful of porting issues that came up and compare libumem's performance to other user-level memory allocators. libumem is still experimental as of this writing.

6.1. Porting Issues

The allocation code (magazine, slab, and vmem) was essentially unchanged; the challenge was to find user-level replacements for the kernel functionality on which it relies, and to accommodate the limitations and interface requirements of user-level library code.

- **CPU ID.** The kernel uses the CPU ID, which can be determined in just a few instructions, to index into a cache's `cache_cpu[]` array. There is no equivalent of CPU ID in the thread library; we need one.* For the prototype we just hashed on the thread ID, which is available cheaply in `libthread`.
- **Memory Pressure.** In the kernel, the VM system invokes `kmem_reap()` when system-wide free memory runs low. There is no equivalent concept in userland. In libumem we check the depot working set size whenever we access the depot and return any excess to the slab layer.
- **Supporting `malloc(3C)` and `free(3C)`.** To implement `malloc()` and `free()` we create a set of about 30 fixed-size object caches to handle small-to-medium `malloc()` requests. We use `malloc()`'s `size` argument to index into a table to locate the nearest cache, e.g. `malloc(350)` goes to the `umem_alloc_384` cache. For larger allocations we use the VM system directly, i.e. `sbrk(2)` or `mmap(2)`. We prepend an 8-byte boundary tag to each buffer so we can determine its size in `free()`.
- **Initialization.** The cost of initializing the kernel memory allocator is trivial compared to the cost of booting, but the cost of initializing libumem is not entirely trivial compared to the cost of `exec(2)`, primarily because libumem must create the 30 standard caches that support `malloc/free`. We therefore create these caches lazily (on demand).

**Our game plan is to make the kernel and thread library cooperate, so that whenever the kernel dispatches a thread to a different CPU, it stores the new CPU ID in the user-level thread structure.

6.2. Performance

A complete analysis of user-level memory allocators is beyond the scope of this paper, so we compared libumem only to the strongest competition:

- the Hoard allocator [Berger00], which appears to be the current best-of-breed among scalable user-level memory allocators;
- ptmalloc [Gloster01], a widely used multithreaded malloc used in the GNU C library;
- the Solaris mtmalloc library.

We also benchmarked the Solaris C library's malloc [Sleator85] to establish a single-threaded baseline.

During our measurements we found several serious scalability problems with the Solaris mtmalloc library. mtmalloc creates per-CPU power-of-two freelists for sizes up to 64K, but its algorithm for selecting a freelist was simply round-robin; thus its workload was

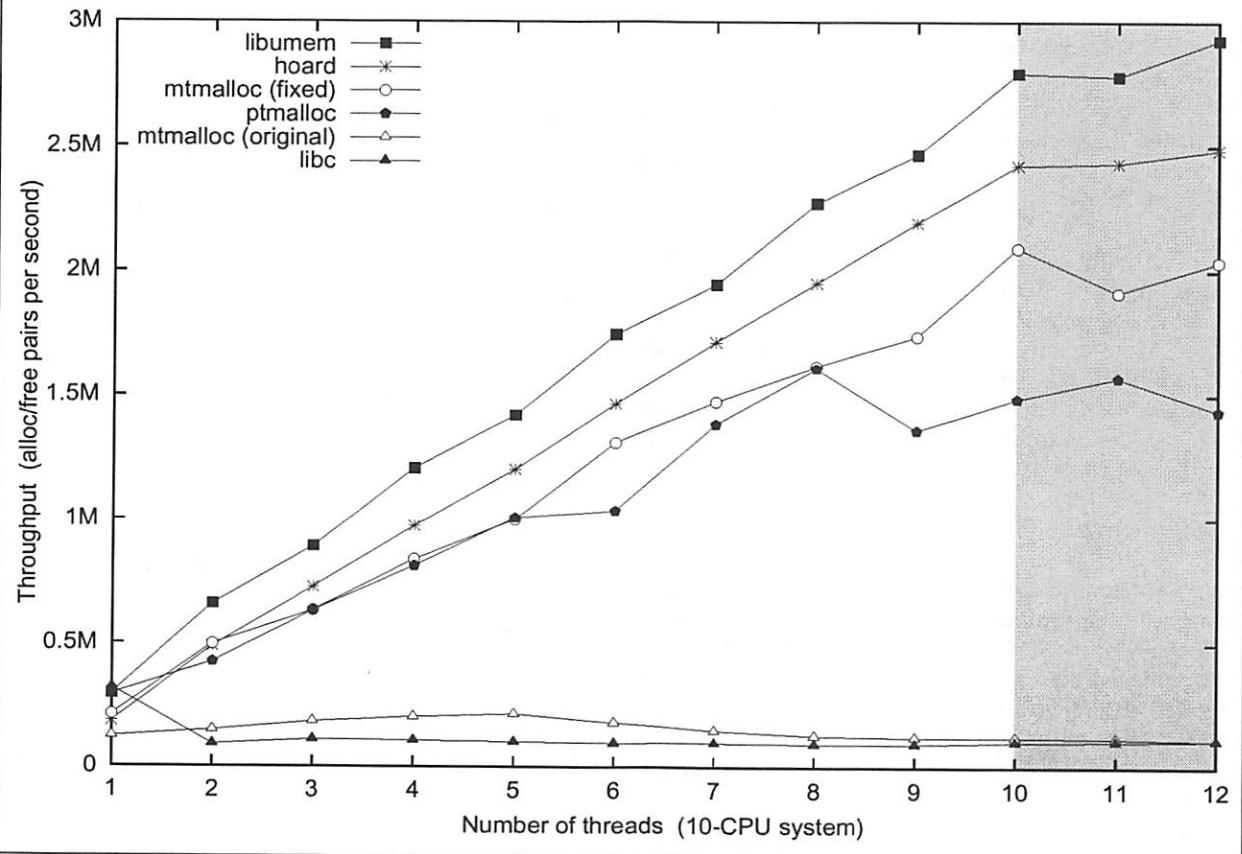
merely fanned out, not made CPU-local. Moreover, the round-robin index was itself a global variable, so frequent increments by all CPUs caused severe contention for its cache line. We also found that mtmalloc's per-CPU data structures were not suitably padded and aligned to cache line boundaries to prevent false sharing, as discussed in §3.6.

We fixed mtmalloc to select a per-CPU freelist by thread ID hashing as in libumem, and we padded and aligned its per-CPU data structures. These changes improved the scalability of mtmalloc dramatically, making it competitive with Hoard and libumem.

We measured the allocators' scalability on a 10-CPU E4000 using the methods described in §3.8. Figure 6.2 shows that libc's malloc and the original mtmalloc perform abysmally as the number of threads increases. ptmalloc provides good scalability up to 8 CPUs, but appears not to scale beyond that. By contrast, libumem, Hoard, and the fixed mtmalloc all show linear scaling. Only the slopes differ, with libumem being the fastest.

Figure 6.2: malloc/free Performance

Note: the shaded area indicates data points where the number of threads exceeds the number of CPUs; all curves necessarily flatten at that point. An allocator with linear scaling should be linear up to the shaded area, then flat.



7. Conclusions

The enduring lesson from our experience with the slab allocator is that it is essential to create excellent core services. It may seem strange at first, but core services are often the most neglected.

People working on a particular performance problem such as web server performance typically focus on a specific goal like better SPECweb99 numbers. If profiling data suggests that a core system service is one of the top five problems, our hypothetical SPECweb99 performance team is more likely to find a quick-and-dirty way to avoid that service than to embark on a major detour from their primary task and redesign the offending subsystem. This is how we ended up with over 30 special-purpose allocators before the advent of vmem.

Such quick-and-dirty solutions, while adequate at the time, do not advance operating system technology. Quite the opposite: they make the system more complex, less maintainable, and leave behind a mess of ticking time bombs that will eventually have to be dealt with. None of our 30 special-purpose allocators, for example, had anything like a magazine layer; thus every one of them was a scalability problem in waiting. (In fact, some were no longer waiting.)

Before 1994, Solaris kernel engineers avoided the memory allocator because it was known to be slow. Now, by contrast, our engineers actively seek ways to use the allocator because it is known to be fast and scalable. They also know that the allocator provides extensive statistics and debugging support, which makes whatever they're doing that much easier.

We currently use the allocator to manage ordinary kernel memory, virtual memory, DMA, minor device numbers, System V semaphores, thread stacks and task IDs. More creative uses are currently in the works, including using the allocator to manage pools of worker threads – the idea being that the depot working set provides an effective algorithm to manage the size of the thread pool. And in the near future, libumem will bring all of this technology to user-level applications and libraries.

We've demonstrated that magazines and vmem have improved performance on real-world system-level benchmarks by 50% or more. But equally important, we achieved these gains by investing in a core system service (resource allocation) that many other project teams have built on. Investing in core services is critical to maintaining and evolving a fast, reliable operating system.

Acknowledgments

We would like to thank:

- Bruce Curtis, Denis Sheahan, Peter Swain, Randy Taylor, Sunay Tripathi, and Yufei Zhu for system-level performance measurements;
- Bryan Cantrill, Dan Price and Mike Shapiro for creating an excellent suite of debugging tools for the slab and vmem allocators, now available in Solaris 8 as part of mdb(1);
- Mohit Aron, Cathy Bonwick, Bryan Cantrill, Roger Faulkner, Dave Powell, Jonathan Shapiro, Mike Shapiro, Bart Smaalders, Bill Sommerfeld, and Mike Sullivan for many helpful comments on draft versions of the paper.

References

Magazines and vmem are part of Solaris 8. The source is available for free download at www.sun.com.

For general background, [Wilson95] provides an extensive survey of memory allocation techniques. In addition, the references in [Berger00], [Bonwick94], and [Wilson95] list dozens of excellent papers on memory allocation.

[Berger00] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, Paul R. Wilson. *Hoard: A Scalable Memory Allocator for Multithreaded Applications*. ASPLOS-IX, Cambridge, MA, November 2000. Available at <http://www.hoard.org>.

[BIRD01] BIRD Programmer's Documentation. Available at <http://bird.network.cz>.

[Bonwick94] Jeff Bonwick. *The Slab Allocator: An Object-Caching Kernel Memory Allocator*. Summer 1994 Usenix Conference, pp. 87–98. Available at <http://www.usenix.org>.

[Bovet00] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. Prentice Hall, 2000.

[Denning68] Peter J. Denning. *The Working Set Model for Program Behaviour*. CACM 11(5), 1968, pp. 323–333.

[FreeBSD01] The FreeBSD source code. Available at <http://www.freebsd.org>.

- [Gloeger01]** Source code and documentation for ptmalloc are available on Wolfram Gloer's home page at <http://www.malloc.de>.
- [Johnstone97]** Mark S. Johnstone and Paul R. Wilson. *The Memory Fragmentation Problem: Solved?* ISMM'98 Proceedings of the ACM SIGPLAN International Symposium on Memory Management, pp. 26–36. Available at <ftp://ftp.dcs.gla.ac.uk/pub/drastic/gc/wilson.ps>.
- [Khanna92]** Sandeep Khanna, Michael Sebree and John Zolnowski. *Realtime Scheduling in SunOS 5.0*. Winter 1992 USENIX Conference.
- [Knuth73]** Donald Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison Wesley, 1973.
- [Linux01]** The Linux source code. Available at <http://www.linux.org>.
- [Mauro00]** Jim Mauro and Richard McDougall. *Solaris Internals: Core Kernel Architecture*. Prentice Hall, 2000.
- [McKenney93]** Paul E. McKenney and Jack Slingwine. *Efficient Kernel Memory Allocation on Shared-Memory Multiprocessors*. Proceedings of the Winter 1993 Usenix Conference, pp. 295–305. Available at <http://www.usenix.org>.
- [Nemesis01]** The Nemesis source code. Available at <http://nemesis.sourceforge.net>.
- [NetBSD01]** The NetBSD source code. Available at <http://www.netbsd.org>.
- [OpenBSD01]** The OpenBSD source code. Available at <http://www.openbsd.org>.
- [Perl01]** The Perl source code. Available at <http://www.perl.org>.
- [Shapiro01]** Jonathan Shapiro, personal communication. Information on the EROS operating system is available at <http://www.eros-os.org>.
- [Sleator85]** D. D. Sleator and R. E. Tarjan. *Self-Adjusting Binary Trees*. JACM 1985.
- [SPEC01]** Standard Performance Evaluation Corporation. Available at <http://www.spec.org>.
- [Swain98]** Peter Swain, Softway. Personal communication.
- [Taylor99]** Randy Taylor, Veritas Software. Personal communication.
- [TPC01]** Transaction Processing Council. Available at <http://www.tpc.org>.
- [Vahalia96]** Uresh Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hall, 1996.
- [Weinstock88]** Charles B. Weinstock and William A. Wulf. *QuickFit: An Efficient Algorithm for Heap Storage Allocation*. ACM SIGPLAN Notices, v.23, no. 10, pp. 141–144 (1988).
- [Wilson95]** Paul R. Wilson, Mark S. Johnstone, Michael Neely, David Boles. *Dynamic Storage Allocation: A Survey and Critical Review*. Proceedings of the International Workshop on Memory Management, September 1995. Available at <http://citeseer.nj.nec.com/wilson95dynamic.html>.

Author Information

Jeff Bonwick (bonwick@eng.sun.com) is a Senior Staff Engineer at Sun Microsystems. He works primarily on core kernel services (allocators, lock primitives, timing, filesystems, VM, scalability) and has created several system observability tools such as `kstat(3K)`, `mpstat(1M)` and `lockstat(1M)`. He is currently leading the design and implementation of a new storage architecture for Solaris.

Jonathan Adams (jonathan-adams@ofb.net) is a senior at the California Institute of Technology. He developed libumem during his summer internship at Sun.

Appendix A: Composing Vmem Arenas and Object Caches

In this Appendix we describe all the key steps to get from system boot to creating a complex object cache.

At compile time we statically declare a few vmem arena structures and boundary tags to get us through boot. During boot, the first arena we create is the primordial `heap_arena`, which defines the kernel virtual address range to use for the kernel heap:

```
heap_arena = vmem_create("heap",
    kernelheap, heapsize,      /* base and size of kernel heap */
    PAGESIZE,                 /* unit of currency is one page */
    NULL, NULL, NULL,          /* nothing to import from -- heap is primordial */
    0,                         /* no quantum caching needed */
    VM_SLEEP);                /* OK to wait for memory to create arena */
```

`vmem_create()`, seeing that we're early in boot, uses one of the statically declared arenas to represent the heap, and uses statically declared boundary tags to represent the heap's initial span. Once we have the heap arena, we can create new boundary tags dynamically. For simplicity, we always allocate a whole page of boundary tags at a time: we select a page of heap, map it, divvy it up into boundary tags, use one of those boundary tags to represent the heap page we just allocated, and put the rest on the arena's free boundary tag list.

Next, we create `kmem_va_arena` as a subset of `heap_arena` to provide virtual address caching (via quantum caching) for up to 8 pages. Quantum caching improves performance and helps to minimize heap fragmentation, as we saw in §4.4.5. `kmem_va_arena` uses `vmem_alloc()` and `vmem_free()` to import from `heap_arena`:

```
kmem_va_arena = vmem_create("kmem_va",
    NULL, 0,                      /* no initial span; we import everything */
    PAGESIZE,                     /* unit of currency is one page */
    vmem_alloc,                   /* import allocation function */
    vmem_free,                    /* import free function */
    heap_arena,                   /* import vmem source */
    8 * PAGESIZE,                 /* quantum caching for up to 8 pages */
    VM_SLEEP);                  /* OK to wait for memory to create arena */
```

Finally, we create `kmem_default_arena`, the backing store for most object caches. Its import function, `segkmem_alloc()`, invokes `vmem_alloc()` to get virtual addresses and then backs them with physical pages:

```
kmem_default_arena = vmem_create("kmem_default",
    NULL, 0,                      /* no initial span; we import everything */
    PAGESIZE,                     /* unit of currency is one page */
    segkmem_alloc,                /* import allocation function */
    segkmem_free,                 /* import free function */
    kmem_va_arena,                /* import vmem source */
    0,                           /* no quantum caching needed */
    VM_SLEEP);                  /* OK to wait for memory to create arena */
```

At this point we have a simple page-level allocator: to get three pages of mapped kernel heap, we could call `vmem_alloc(kmem_default_arena, 3 * PAGESIZE, VM_SLEEP)` directly. In fact, this is precisely how the slab allocator gets memory for new slabs. Finally, the kernel's various subsystems create their object caches. For example, the UFS filesystem creates its inode cache:

```
inode_cache = kmem_cache_create("ufs_inode_cache",
    sizeof (struct inode),        /* object size */
    0,                            /* use allocator's default alignment */
    ufs_inode_cache_constructor,  /* inode constructor */
    ufs_inode_cache_destructor,   /* inode destructor */
    ufs_inode_cache_reclaim,     /* inode reclaim */
    NULL,                         /* argument to above funcs */
    NULL,                         /* implies kmem_default_arena */
    0);                          /* no special flags */
```

Appendix B: Vmem Arenas and Object Caches in Solaris 8

The data on this page was obtained by running the `:kmemstat` command under `mdb(1)` on a large Solaris 8 server. It was substantially trimmed to fit the page.

The (shortened) list of all vmem arenas appears below; the (shortened) list of all object caches appears to the right. Shaded regions show the connection between vmem arenas and their quantum caches. [Note: vmem names its quantum caches by appending the object size to the arena name, e.g. the 8K quantum cache for `kmem_va` is named `kmem_va_8192`.]

Arena names are indented in the table below to indicate their importing relationships. For example, `kmem_default` imports virtual addresses from `kmem_va`, which in turn imports virtual addresses from `heap`.

The allocation statistics demonstrate the efficacy of quantum caching. At the time of this snapshot there had been over a million allocations for `sbus0_dvma` (1.18 million 8K allocations, as shown in the total allocation column for `sbus0_dvma_8192`; 309,600 16K allocations, and so on). All of this activity resulted in just 14 segment list allocations. Everything else was handled by the quantum caches.

vmem arena name	memory in use	memory imported	total allocs
heap	650231808	0	20569
vmem_seg	9158656	9158656	1118
vmem_vmem	128656	81920	81
kmem_internal	28581888	28581888	4339
kmem_cache	667392	974848	334
kmem_log	1970976	1974272	6
kmem_oversize	30067072	30400512	3616
mod_sysfile	115	8192	4
kmem_va	557580288	557580288	2494
kmem_default	557137920	557137920	110966
little_endian	0	0	0
bp_map	18350080	18350080	7617
ksyms	685077	761856	125
heap32	1916928	0	58
id32	16384	16384	2
module_text	2325080	786432	120
module_data	368762	1032192	165
promplat	0	0	15
segkp	449314816	0	3749
taskid_space	3	0	4
sbus0_dvma	3407872	0	14
...			
sbus7_dvma	2097152	0	12
ip_minor	256	0	4
ptms_minor	1	0	1

object cache name	obj size	objs in use	total allocs
kmem_magazine_1	16	1923	3903
kmem_magazine_3	32	6818	56014
kmem_magazine_7	64	29898	37634
kmem_magazine_15	128	26210	27237
kmem_magazine_31	256	4662	8381
kmem_magazine_47	384	4149	7003
kmem_magazine_63	512	0	3018
kmem_magazine_95	768	1841	3182
kmem_magazine_143	1152	6655	6655
kmem_slab_cache	56	29212	31594
kmem_bufctl_cache	32	222752	249720
kmem_va_8192	8192	67772	111495
kmem_va_16384	16384	77	77
kmem_va_24576	24576	28	29
kmem_va_32768	32768	0	0
kmem_va_40960	40960	0	0
kmem_va_49152	49152	0	0
kmem_va_57344	57344	0	0
kmem_va_65536	65536	0	0
kmem_alloc_8	8	51283	57609715
kmem_alloc_16	16	4185	19065575
kmem_alloc_24	24	2479	76864949
...			
kmem_alloc_16384	16384	52	162
streams_mblk	64	128834	142921
streams_dbblk_8	128	8	464076
streams_dbblk_40	160	205	10722289
streams_dbblk_72	192	302	201275
...			
streams_dbblk_12136	12256	0	0
streams_dbblk_esb	120	0	3
id32_cache	8	1888	1888
bp_map_16384	16384	0	6553071
bp_map_32768	32768	0	2722
bp_map_49152	49152	0	292
bp_map_65536	65536	0	21
bp_map_81920	81920	0	768
bp_map_98304	98304	0	995
bp_map_114688	114688	0	5
bp_map_131072	131072	0	99
sfmmuid_cache	48	35	7617426
sfmmu8_cache	312	358161	389921
sfmmu1_cache	88	126878	138258
seg_cache	64	1098	134076345
segkp_8192	8192	0	0
segkp_16384	16384	79	79
segkp_24576	24576	722	845690
segkp_32768	32768	0	0
segkp_40960	40960	0	1213
thread_cache	672	229	4027805
lwp_cache	880	229	1260382
turnstile_cache	64	749	3920308
cred_cache	96	8	866335
dnlc_space_cache	24	819	565894
file_cache	56	307	46876583
queue_cache	608	604	991955
syncq_cache	160	18	112
as_cache	144	34	7727219
anon_cache	48	4455	112122999
anonmap_cache	56	676	60732684
segvn_cache	96	1096	121023992
snode_cache	256	379	1183612
ufs_inode_cache	480	23782	3156269
sbus0_dvma_8192	8192	66	1180296
sbus0_dvma_16384	16384	2	309600
sbus0_dvma_24576	24576	2	13665
sbus0_dvma_32768	32768	0	154246
sbus0_dvma_40960	40960	0	0
sbus0_dvma_49152	49152	0	0
sbus0_dvma_57344	57344	0	0
sbus0_dvma_65536	65536	0	0
fas0_cache	256	26	21148
fas1_cache	256	0	15
fas2_cache	256	0	15
fas3_cache	256	0	15
sock_cache	432	45	234
sock_unix_cache	432	0	8
ip_minor_1	1	116	549
process_cache	2688	37	3987768
fnode_cache	264	6	55
pipe_cache	496	8	545626
authkern_cache	72	0	312
authloopback_cache	72	0	232
authdes_cache_handle	72	0	0
rnode_cache	656	3	15
nfs_access_cache	40	2	20
client_handle_cache	32	4	4
pty_map	48	1	1

Measuring Thin-Client Performance Using Slow-Motion Benchmarking

S. Jae Yang, Jason Nieh, and Naomi Novik

Department of Computer Science

Columbia University

{sy180, nieh, nn80}@cs.columbia.edu

Abstract

Modern thin-client systems are designed to provide the same graphical interfaces and applications available on traditional desktop computers while centralizing administration and allowing more efficient use of computing resources. Despite the rapidly increasing popularity of these client-server systems, there are few reliable analyses of their performance. Industry standard benchmark techniques commonly used for measuring desktop system performance are ill-suited for measuring the performance of thin-client systems because these benchmarks only measure application performance on the server, not the actual user-perceived performance on the client.

To address this problem, we have developed *slow-motion benchmarking*, a new measurement technique for evaluating thin-client systems. In slow-motion benchmarking, performance is measured by capturing network packet traces between a thin client and its respective server during the execution of a slow-motion version of a standard application benchmark. These results can then be used either independently or in conjunction with standard benchmark results to yield an accurate and objective measure of the performance of thin-client systems.

We have demonstrated the effectiveness of slow-motion benchmarking by using this technique to measure the performance of several popular thin-client systems in various network environments on web and multimedia workloads. Our results show that slow-motion benchmarking resolves the problems with using standard benchmarks on thin-client systems and is an accurate tool for analyzing the performance of these systems.

1 Introduction

The rising cost of support and maintenance for desktop systems has fueled a growing interest in thin-client computing. Modern thin-client systems are designed to provide the same graphical interfaces and applications available on desktop systems while centralizing computing work on powerful servers to reduce administration costs and make more efficient use of shared computing resources.

While the term “thin-client computing” has been used to refer to a variety of different client-server computing architectures, the primary feature common to most thin-client systems is that all application logic is executed on the server, not on the client. The user interacts with a lightweight client that is generally responsible only for handling user input and output, such as receiving screen display updates and sending user input back to the server over a network connection. Unlike older client-server architectures such as X [7], many modern thin-client systems even run the window system on the server. As a result, the client generally does not need many resources, thus requiring fewer upgrades, and can

have a very simple configuration, reducing support costs. Because of the potential cost benefits of thin-client computing, a wide range of thin-client platforms has been developed. Some application service providers (ASPs) are even offering thin-client service over wide area networks such as the Internet [8, 10, 22].

The growing popularity of thin-client systems makes it important to develop techniques for analyzing their performance, to assess the general feasibility of the thin-client computing model, and to compare different thin-client platforms. However, because thin-client platforms are designed and used very differently from traditional desktop systems, quantifying and measuring their performance effectively is difficult. Standard benchmarks for desktop system performance cannot be relied upon to provide accurate results when used to measure thin-client systems. Because benchmark applications running in a thin-client system are executed on the server, these benchmarks effectively only measure the server’s performance and do not accurately represent the user’s experience at the client-side of the system. To make matters more difficult, many of these systems are proprietary and closed-

source, making it difficult to instrument them and obtain accurate results.

To address this problem, we introduce *slow-motion benchmarking*, a new measurement technique for evaluating thin-client systems. In slow-motion benchmarking, performance is measured by capturing network packet traces between a thin client and its respective server during the execution of a slow-motion version of a standard application benchmark. These results can then be used either independently or in conjunction with standard benchmark results to yield an accurate and objective measure of user-perceived performance for applications running over thin-client systems.

To demonstrate the accuracy of this technique, we have used slow-motion benchmarking to measure the performance of four popular thin-client systems on both web and multimedia applications. The thin-client systems evaluated were Microsoft Terminal Services [17], Citrix MetaFrame [4], AT&T VNC [32], and Sun Ray [30]. We measured the performance of these systems over various network access bandwidths, ranging from ISDN up to LAN network environments. Our results illustrate the performance differences between these thin-client systems and demonstrate the effectiveness of slow-motion benchmarking as a tool for analyzing thin-client system performance. We have also compared our results to the results obtained using standard application benchmarking approaches. These comparisons illustrate the limitations of previous thin-client benchmarking efforts based on widely-used industry standard benchmarks.

The rest of this paper is organized as follows. Section 2 describes how thin-client systems operate in further detail and then explains the difficulties inherent in measuring the performance of thin-client platforms with standard benchmarks. Section 3 presents the slow-motion benchmarking technique and discusses how it can be used to measure thin-client performance. Section 4 presents examples of benchmarks that we have modified for slow-motion benchmarking and describes how these may be used to evaluate thin-client systems. Section 5 compares slow-motion benchmarking to standard benchmarking by presenting experimental results that quantify the performance of popular thin-client systems on web and multimedia application workloads over different network bandwidths. Section 6 discusses related work. Finally, we summarize our conclusions and discuss opportunities for future work.

2 Measuring Thin-Client Performance

To provide sufficient background for discussing the issues in measuring thin-client performance, we first describe in further detail how thin-client systems operate. In this paper, we focus on thin-client systems in which a user's complete desktop computing environment, including both application logic and the windowing system, is entirely run on the server. This is the architecture underlying most modern systems referred to as thin-clients, such as Citrix MetaFrame and Microsoft Terminal Services. One of its primary advantages is that existing applications for standalone systems can be used in such systems without modification.

In this type of architecture, the two main components are a client application that executes on a user's local desktop machine and a server application that executes on a remote system. The end user's machine can be a hardware device designed specifically to run the client application or simply a low-end personal computer. The remote server machine typically runs a standard server operating system, and the client and server communicate across a network connection between the desktop and server. The client sends input data across the network to the server, and the server, after executing application logic based on the user input, returns display updates encoded using a platform-specific remote display protocol. The updates may be encoded as high-level graphics drawing commands, or simply compressed pixel data. For instance, Citrix MetaFrame encodes display updates as graphics drawing commands while VNC encodes display updates as compressed pixel data.

To improve remote display performance, especially in environments where network bandwidth is limited, thin-client systems often employ three optimization techniques: compression, caching, and merging. With compression, algorithms such as zlib or run-length encoding can be applied to display updates to reduce bandwidth requirements with only limited additional processing overhead. With caching, a client cache can be used to store display elements such as fonts and bitmaps, so the client can obtain some frequently used display elements locally rather than repeatedly requesting them from the server. With merging, display updates are queued on the server and then merged before they are sent to the client. If two updates change the same screen region, merging will suppress the older update and only send the more recent update to the client. In merging, display updates are sent asynchronously with respect to application execution, decoupling application rendering of visual output on the

server from the actual display of the output on the client. Depending on the merging policy and the network speed, there may be a significant time lapse between the time application rendering occurs on the server and the time the actual display occurs on the client. These optimizations, particularly merging, can make analyzing thin-client performance difficult.

The performance of a thin-client system should be judged by what the user experiences on the client. There are two main problems encountered when trying to analyze the performance of thin-client systems. The first problem is how to correctly measure performance of the overall system rather than simply the server's performance. The second, more subtle problem is how to objectively measure the resulting display quality, particularly given the display update optimizations that may be employed. Three methods that have been used to measure thin-client system performance are internal application measurements with standard benchmarks, client instrumentation, and network monitoring. We discuss each of these methods below and their limitations.

The first approach, commonly used for traditional desktop systems, is to simply run a standard benchmark on the system. For instance, a video playback application could be run that reports the frame rate as a measure of performance. However, this does not provide an accurate measure of overall thin-client performance because the application programs are executed entirely on the server. Since application execution is often decoupled from client display, the results reported using internal application measurements might not be an accurate reflection of user-perceived performance on the client. The benchmark program often runs on the server irrespective of the status of the display on the client. A video playback application, for example, would measure the frame rate as rendered on the server, but if many of the frames did not reach the client, the frame rate reported by the benchmark would give an overly optimistic view of performance.

A second, more accurate measurement method would be to directly instrument the client. If appropriate tracing mechanisms could be inserted into the thin-client system to log input and output display events on the client, very detailed measurements of thin-client performance could be performed. However, many thin-client systems are quite complex and instrumenting them effectively would not be easy. Furthermore, the information that could be obtained within these systems would still not provide a direct measure of user-perceived display quality. Running a video playback

application, for example, would result in thousands of display updates. One would still be left with the problem of how to determine how those display updates translate into actual video frames to determine how many video frames were delivered to the client. A more practical problem is that many of the most popular thin-client systems, such as Citrix MetaFrame, Windows Terminal Services, SCO Tarantella, and Sun Ray, are all proprietary, closed systems. Even the specification of the remote display protocol used in these systems is not available.

A third measurement method is network monitoring. While just measuring application performance on the server can be inaccurate and direct thin-client instrumentation is often not possible, monitoring network activity between the client and server during the execution of an application can give us a closer approximation to the user-perceived performance of that application on the client-side. We can measure the latency of operations such as web page downloads as the time between the start and end of client-server communication in response to those operations. However, while this method enables us to more accurately measure the latency of display updates for such operations, we are still left with the question of determining the resulting display quality. Fast performance is naturally an important metric, but it is insufficient when considered in isolation. The user's interactive experience is equally determined by the visual quality of the updates, and in many cases platforms may achieve high speed screen refresh rates by discarding data, which does not necessarily lead to good interactive performance from the user's perspective.

In particular, thin-client systems that use display update merging may drop interim display updates if the client cannot keep up with the rate of display updates generated on the server. A thin-client platform that uses this kind of policy will appear to perform well on benchmarks measuring only the latency of display updates, even at very low bandwidths, because it will simply discard whatever data cannot be transmitted quickly enough. This problem is exacerbated by most standard benchmarks for measuring graphical performance, which typically execute a rapid sequence of tasks with frequent display changes. For instance, the standard industry benchmark i-Bench [35] from Ziff-Davis measures performance on web applications with a rapid-fire series of web page downloads, each page triggering the download of the next page when complete. This technique works for traditional desktop systems, but on a thin-client system, the server can end up finishing one page download and starting the next

long before the client has finished displaying the first page. The server may even stop sending the display updates associated with the first page and send the client on to the second regardless of whether the client has finished its display.

Monitoring the amount of data transferred for display updates at different network bandwidths can help to determine when display update merging is occurring, but other problems remain. For one, simple network monitoring cannot quantify the amount of display updates still being discarded at the highest available bandwidths. In addition, because each thin-client platform encodes display updates in its own proprietary protocol, network monitoring alone cannot determine whether the thin-client platforms are all transmitting the same overall visual display data, making it impossible to effectively compare the performance of the platforms to each other. Monitoring network traffic at the client is an improvement over server-side application measurements, but we still cannot correctly measure overall performance in a way that accounts for both system response time and display quality.

3 Slow-Motion Benchmarking

To provide a more effective method for measuring thin-client performance, we introduce slow-motion benchmarking. In slow-motion benchmarking, we use network packet traces to monitor the latency and data transferred between the client and the server, but we alter the benchmark application by introducing delays between the separate visual components of that benchmark, such as web pages or video frames, so that the display update for each component is fully completed on the client before the server begins processing the next one.

Figure 1 and Figure 2 illustrate the difference in network traffic between standard and slow-motion versions of an i-Bench web benchmark that downloads a sequence of web pages. The benchmark is described in Section 4.1. The data presented is from

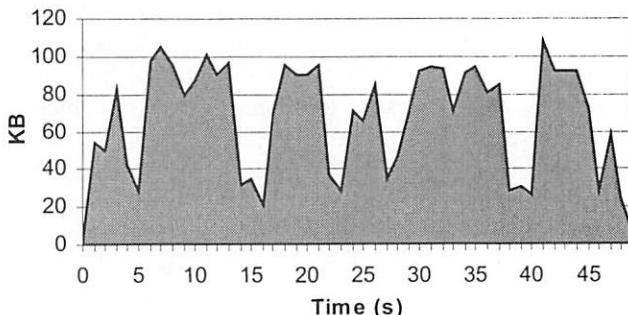


Figure 1: KB transferred at one-second intervals during a sequence of web page downloads with no delays under Citrix MetaFrame at 100 Mbps.

measurements for one thin-client platform, Citrix MetaFrame, with a 100 Mbps network connection between client and server. In the standard benchmark with no delays, the pages run together and cannot be separately identified, even at this high network bandwidth. In the slow-motion version of the benchmark with delays inserted between each page download, the display update data for each separate page is clearly demarcated.

With slow-motion benchmarking, we process the network packet traces and use these gaps of idle time between components to break up the results on a per-component basis. This allows us to obtain the latency and data transferred for each visual component separately. We can then obtain overall results by taking the sum of these results. The amount of the delay used between visual components depends on the application workload and platform being tested. The necessary length of delay can be determined by monitoring the network traffic and making the delays long enough to achieve a clearly demarcated period between all the visual components where client-server communication drops to the idle level for that platform. This ensures that each visual component is discrete and generated completely.

Slow-motion benchmarking has many advantages. First and most importantly, it ensures that display events reliably complete on the client so that capturing them using network monitoring provides an accurate measure of system performance. Slow-motion benchmarking ensures that clients display all visual components in the same sequence, providing a common foundation for making comparisons among thin-client systems.

Second, slow-motion benchmarking does not require any invasive modification of thin-client systems, which is difficult even for open-source systems such as VNC and nearly impossible for proprietary systems. Additionally, since no invasive instrumentation is required, slow-motion benchmarking does not result in any additional performance overhead for the thin-client

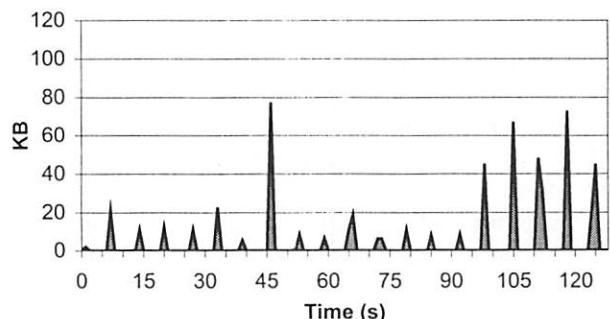


Figure 2: KB transferred at one-second intervals during a slow-motion version of the same web page sequence. For visual clarity, only a subset of the full 109-page sequence represented by Figure 1 is shown here.

system being measured.

Third, slow-motion benchmarking provides an effective way to determine when display updates are being discarded. Since the modified benchmarks run at a slower rate, the resource requirements are reduced, which in turn reduces the likelihood that display updates will be discarded. We can then compare the amount of data transferred for the standard and slow-motion versions of a given application benchmark to determine whether the display updates are being discarded even at the highest network bandwidths. In Section 4.2, we show how this is particularly useful for measuring the performance of video applications on thin-client systems.

Fourth, slow-motion benchmarking is actually closer to standard user behavior for some applications, notably interactive activities such as web browsing. Unlike the behavior of web benchmarks such as those in i-Bench, most users do not click through a hundred web pages in near-instantaneous succession; they wait for a page to visually complete loading before they move on to the next one.

Finally, the delays introduced with slow-motion benchmarking allow us to obtain results with finer granularity at the level of individual visual components. This is very useful for studying the effects of different thin-client remote display mechanisms on different kinds of display data. For instance, some platforms may prove to be better than others at downloading text-only web pages, while others may be superior at graphics-heavy pages. These more detailed results enable us to make better judgments about the various design choices made in each thin-client platform.

In designing slow-motion benchmarking, we made three assumptions. First, we assumed that introducing the delays between visual components would not inherently change the type or amount of data that should be transferred between client and server in a thin-client system. As far as we know, none of the thin-client platforms fundamentally alter the way they encode and send updates based on the amount of time between visual components.

Second, we assumed that there would not be extraneous packets in the data stream that would change our measurements. In particular, we assumed that the delays introduced between visual components would be directly reflected in noticeable gaps in the network packet traces captured. For almost all thin-client platforms and application benchmarks that we measured, there were no packets during the idle periods. However, on certain platforms such as Sun

Ray, some small packets were transmitted even during idle periods, possibly to make sure that the connection had not been lost. However, we found that a judicious filtering process based on the volume of idle-time data allowed us to successfully distinguish the data transferred for the pages from the overhead.

Third, we assumed that monitoring network traffic generated by the slow-motion benchmarks was a valid measure of overall performance. Network monitoring allows us to completely measure network and server latency, but may not provide a complete end-to-end measure of client latency. Network monitoring does account for all client latency that occurs between the first and last packet generated for a visual component. However, this technique does not account for any client processing time required for displaying a visual component that occurs before the first network packet or after the last network packet for that component. The impact of this limitation depends on the importance of client latency to the overall performance. If, as we expected, network and server latency were the dominant factors in overall performance, the additional client latency would not be significant. However, if the client latency were a significant component of overall performance, network monitoring might not completely measure end-to-end performance. Client latency would typically be large if the client were heavily loaded. We therefore compensated for this limitation by monitoring client load with standard system monitoring tools such as perfmon and sysload to check whether the client was heavily loaded. We found that client load was generally not an issue and that the network was typically the primary performance bottleneck. The one instance in which this was not the case was for VNC running over high network bandwidths. However as we discuss in Section 5.2.2, we also instrumented VNC directly and found that the packet traces accounted for all client latency for this platform.

4 Examples of Slow-Motion Benchmarks

To illustrate how slow-motion benchmarking can be used in practice, we describe two examples of how application benchmarks can be modified to use slow-motion benchmarking. The two examples are taken from the Ziff-Davis i-Bench benchmark suite version 1.5 [35], a benchmarking tool that has been used by numerous computer companies and Ziff-Davis Labs for measuring the performance of a variety of desktop and thin-client systems. The i-Bench benchmarks used were the Web Text Page Load and MPEG1 Video benchmarks, which can be used to measure system

performance on web-based and multimedia applications.

4.1 Web Text Page Load Benchmark

The Web Text Page Load benchmark measures the total time required to download a Javascript-controlled sequence of web pages from a web server. In the unmodified benchmark, the web pages are downloaded one after another without user input using a Javascript-enabled web browser, such as Netscape or Internet Explorer. Each page contains a small script initiated by the Javascript onLoad handler that immediately begins downloading the next page once the web browser has finished downloading the current one. The benchmark cycles through a set of 54 unique web pages twice and then reports the elapsed time in milliseconds on a separate web page. Including the final results page, a total of 109 web pages are downloaded during this test. The 54 pages contain both text and bitmap graphics, with some pages containing more text while others contain more graphics, with some common graphical elements included on each page.

There are two problems with using the unmodified Web Text Page Load benchmark for measuring thin-client performance. The first problem is that since the benchmark would execute in the web browser on the server of a thin-client system, the Javascript handler may indicate that a given web page has been completely downloaded on the server and move on to the next page while the given page has not yet been completely displayed on the client. The second problem is that since the benchmark only provides an overall latency measure for downloading an entire sequence of web pages, it does not provide an accurate measure of per page download latencies and how performance may vary with page content.

We can address these problems by applying slow-motion benchmarking as follows. The visual components of this benchmark are the individual web pages. To break up the benchmark into its separate components, we can simply alter the Javascript code used to load the successor page so that it introduces delays of several seconds between web pages. The delay should be long enough to ensure that the thin client receives and displays each page completely before the thin server began downloading the next web page. Furthermore, the delays should be long enough to ensure that there is no temporal overlap in transferring the data belonging to two consecutive pages. A longer delay might be required in systems with lower network bandwidth between client and server.

By using a slow-motion version of the Web Text Page Load benchmark modified along these lines, we can ensure that each web page is completely displayed on the client and measure performance on a per-page basis. As a result, we can conduct performance comparisons of different thin-client systems knowing that each of them is correctly displaying the same set of web pages. In addition, we can use the per-page measurements to determine how page download latency and the amount of data transferred varies with page content. By performing these measurements with various network bandwidths between client and server, we can determine how the response time of a thin-client system varies with network access bandwidth. Given a model of how frequently users move between web pages, we can use the slow-motion benchmarking measurements to determine whether a thin-client system can provide sufficient response time for a given network connection to ensure a good web browsing experience.

4.2 MPEG1 Video Benchmark

The MPEG1 Video benchmark measures the total time required to playback an MPEG1 video file containing a mix of news and entertainment programming. The video is a 34.75 second clip that consists of 834 352x240 pixel frames with an ideal frame rate of 24 frames per second (fps). The ideal frame rate is the rate a video player would use for playing the video file in the absence of resource limitations that would make this impossible. The total size of the video file is 5.11 MB. In running the video benchmark on a thin-client system, the video player would run on the server and decode and render the MPEG1 video on the server. The remote display protocol of the thin-client system would then send the resulting display updates to the client. Note that unlike streaming MPEG media systems that transmit MPEG video to the client for decoding, thin-client systems first decode the video on the server and then transmit display updates using their own remote display protocol.

There are two problems with using the unmodified MPEG1 Video benchmark for measuring thin-client performance. The first problem is that playback time alone is a poor measure of video performance. Some video players discard video frames when the system is not fast enough to decode and display every frame. The second problem is that in thin-client systems, the system itself may also drop video frames by discarding their corresponding display updates when the network between the client and server is congested. The resulting lower video quality is not properly accounted for by either the playback-time metric or the video player's accounting of dropped video frames.

We can address these problems by applying slow-motion benchmarking as follows. In this case, the visual components of the benchmark are the individual video frames. We can isolate these frames simply by reducing the video playback rate of the benchmark. The playback rate should be slow enough to ensure that there is enough time to decode and display each video frame before the next one needs to be processed. Although users would not actually watch video at such a greatly reduced playback rate, the measurements at this reduced playback rate can be used to establish the reference data transfer rate from the thin server to the client that corresponds to a “perfect” playback without discarded video frames. The data transfer rate can be calculated as the total data transferred divided by the total playback time. We can then compare the data transfer rate at the reduced playback rate with the corresponding full playback rate measurements to determine the video quality achieved at full playback rate. More specifically, we can use the following formula as a measure of video quality VQ at a given specified playback rate P:

$$VQ(P) = \frac{\left[\frac{\text{DataTransferred}(P) / \text{PlaybackTime}(P)}{\text{IdealFPS}(P)} \right]}{\left[\frac{\text{DataTransferred(slow-mo)} / \text{PlaybackTime(slow-mo)}}{\text{IdealFPS(slow-mo)}} \right]}$$

For example, suppose playing a video at an ideal 24 fps rate takes half a minute and results in 10 MB of data being transferred while playing the video at a slow-motion ideal 1 fps rate takes 12 minutes and results in 20 MB of data being transferred. Then, based on the above formula, the resulting video quality VQ at 24 fps will be 0.5 or 50%, which is what one would expect since the 24 fps video playback discarded half of the video data. The effectiveness of this formula depends on the platform’s ability to maintain the 1 fps frame rate. In our experiments, all of the platforms closely conformed to the frame rate.

While this metric provides a useful, non-invasive way

to measure video quality, it only accounts for the amount of data discarded and does not account for the fact that some video data may be more important to the overall display quality of a given video sequence than other data. For instance, discarding display updates corresponding to a video frame that looks almost the same as both the previous and next video frames in a sequence would not change the perceived display quality as much as discarding updates corresponding to a video frame that is unlike any of its neighboring frames. At the same time though, as discussed in Section 2, many of the thin-client systems use some form of compression to reduce the data size of their display updates. Compression effectively reduces data size by removing redundant information in the data. If we assume that the amount of unique information in a display update is a measure of its importance, then a compressed display update could be viewed in a rough sense as being scaled according to its importance. In this case, the proposed measure of video quality based on the amount of discarded compressed data effectively accounts for the fact that different display data may be of different importance.

5 Experimental Results

To demonstrate the effectiveness of slow-motion benchmarking, we evaluated four popular thin-client platforms using the unmodified and slow-motion versions of the web and video benchmarks described in Section 4. The platforms we measured were Citrix MetaFrame, Windows Terminal Services, AT&T VNC, and Sun Ray. Section 5.1 describes our experimental design, including the hardware and software testbed we used, the thin-client platform configurations we tested, and the experiments we conducted. Sections 5.2 and 5.3 discuss our measurements and results comparing slow-motion benchmarking against using the standard unmodified benchmarks. The results also contrast the performance of different thin-client systems on web and video applications.

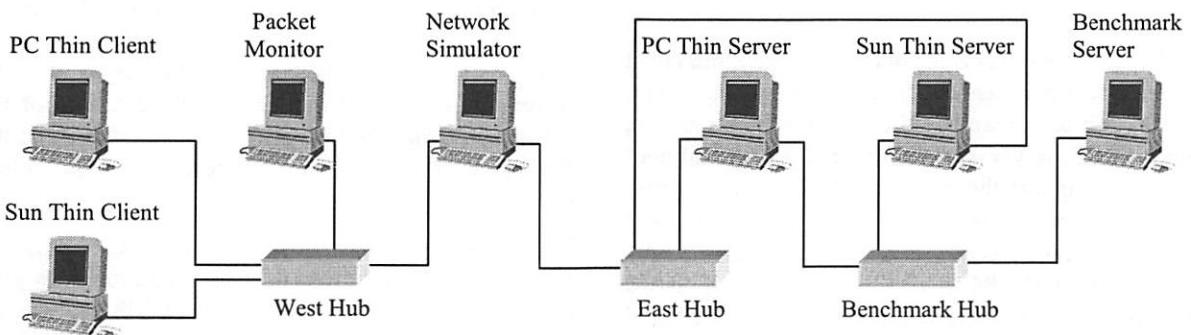


Figure 3: Testbed configuration.

Role / Model	Hardware	OS / Window System	Software
PC Thin Client Micron Client Pro	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 10/100BaseT NIC nVidia Riva TNT graphics adapter, 16 MB SDRAM	MS Win NT 4.0 Workstation SP6 Caldera OpenLinux 2.4, Xfree86 3.3.6, KDE 1.1.2	Citrix ICA Win32 Client MS RDP5 Client VNC Win32 3.3.3r7 Client
Sun Thin Client Sun Ray I	100 MHz Sun uSPARC IIep 8 MB RAM 10/100BaseT NIC ATI Rage 128 graphics adapter	Sun Ray OS	N/A
Packet Monitor Micron Client Pro	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 10/100BaseT NIC	MS Win 2000 Professional	AG Group's Etherpeak 4
Network Simulator Micron Client Pro	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 2 10/100BaseT NICs	MS Win NT 4.0 Server SP6a	Shunra Software The Cloud 1.1
PC Thin Server Micron Client Pro (SPEC95 – 17.2 int, 12.9 fp)	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 2 10/100BaseT NICs	MS Win 2000 Advanced Server Caldera OpenLinux 2.4, Xfree86 3.3.6, KDE 1.1.2	Citrix MetaFrame 1.8 MS Win 2000 Terminal Services AT&T VNC 3.3.3r2 for Linux Netscape Communicator 4.72
Sun Thin Server Sun Ultra-10 Creator 3D (SPEC95 – 14.2 int, 16.9 fp)	333 MHz UltraSPARC III 384 MB RAM 9 GB Disk 2 10/100BaseT NICs	Sun Solaris 7 Generic 106541-08, OpenWindows 3.6.1, CDE 1.3.5	Sun Ray Server 1.2_10.d Beta Netscape Communicator 4.72
Benchmark Server Micron Client Pro	450 MHz Intel PII 128 MB RAM 14.6 GB Disk 10/100BaseT NIC	MS Win NT 4.0 Server SP6a	Ziff-Davis i-Bench 1.5 MS Internet Information Server
Network Hub Linksys NH1005	3 10/100 5-Port Hubs	N/A	N/A

Table 1: Summary of testbed configuration.

5.1 Experimental Design

5.1.1 Experimental Testbed

Our testbed, shown in Figure 3, consisted of two pairs of client/server systems, a network simulator machine, a packet monitor machine, and a benchmark server. Only one client/server pair was active during any given test. The features of each system are summarized in Table 1, along with the SPEC95 performance numbers for each server system.

The client/server systems included a Sun Ray thin client machine and a Sun server, and a PC client and server. The Sun thin server was used only for Sun Ray testing while the PC server was configured as a dual-boot machine to support the various Windows- and Linux-based thin-client systems. The Sun Ray client was considerably less powerful than the PC client, with only a 100 MHz uSPARC CPU and 8 MB of RAM compared to a 450 MHz Pentium II with 128 MB of RAM in the PC client. However, the large difference in

client processing power was not a factor in our evaluations, as the client systems were not generally heavily loaded during testing.

As shown in Figure 3, the network simulator machine was placed between the thin client and thin server machines to control the bandwidth between them. This simulator ran a software package called The Cloud [29], which allowed us to vary the effective bandwidth between the two network interface cards installed in the system. The thin clients and thin servers were separated from one another on isolated 100 Mbps networks. The server-side network was then connected to one of the network interfaces in the network simulator PC, and the client-side network was connected to the other interface.

To ensure that this simulator did not itself introduce extra delay into our tests, we measured round-trip ping times from the client to the server at 100 Mbps, with and without the simulator inserted between the client

and the server. There were no significant differences and round-trip ping times were roughly 0.6 ms in both cases.

To monitor the client-server network traffic, we used a PC running Etherapeek 4 [1], a software packet monitor that timestamps and records all packet traffic visible to the PC. As shown in Figure 3, we primarily used the packet monitor to observe client-side network traffic. In order to capture all packet traffic being sent in both directions between the thin client and server, we used hubs rather than switches in our testbed. Since traffic going through a hub is broadcast to all other machines connected to the hub, this enabled us to record network traffic between the client and server simply by connecting the packet monitor to the hub that the data was passing through.

A limitation of this network setup is that the hubs are half-duplex, so that traffic cannot be sent through the hub from client to server and from server to client concurrently. Since most data in these thin-client platforms is traveling from the server to the client in any case, it is unlikely that the half-duplex network added significant delay to our experiments.

Other options are possible, each with its disadvantages. One alternative would be to run a packet monitor on the thin client or thin server, but Etherapeek is highly resource-intensive and would undoubtedly adversely affect performance results. Furthermore, in the case of the Sun Ray thin client device, it is not possible to run a packet monitor locally on the client. Another alternative would be to use port-mirroring switches to support full-duplex network connections, but mirroring typically would only allow monitoring of either client to server traffic or vice versa, not both at the same time, as mirroring a duplex port in both directions simultaneously can result in packet loss [6].

Finally, we also had a separate benchmark server, which was used to run our modified version of the web page benchmark described in Section 4.1. To ensure that network traffic from the benchmark server did not interfere with the network connection between thin client and thin server, the benchmark server was connected to the testbed using a separate hub, as shown in Figure 3. Each thin server had two 100 Mbps network interfaces, one connected to the network simulator and through that to the client, the other

connected to the benchmark server on a separate channel.

5.1.2 Thin-Client Platforms

The versions of the four thin-client systems tested are shown in the last column of Table 1. Citrix MetaFrame and Terminal Services were run with Windows 2000 servers while VNC and Sun Ray were run with UNIX servers, Linux and Solaris. It was necessary to use different server operating systems because Terminal Services is part of Windows 2000, VNC performs much better on UNIX than Windows [32], and Sun Ray only works with Solaris. However to minimize system differences across thin-client platforms, all platforms except for Sun Ray used the exact same server hardware and same client OS and hardware.

The thin-client platform configurations used for our experiments are listed in Table 2. To minimize application environment differences, we used common thin-client configuration options and common applications across all platforms whenever possible. Where it was not possible to configure all the platforms in the same way, we generally used default settings for the platforms in question.

For all of our experiments, the video resolution of the thin client was set to 1024x768 resolution with 8-bit color, as this was the lowest common denominator supported by all of the platforms. However, the Sun Ray client was set to 24-bit color, since the Sun Ray display protocol is based on a 24-bit color encoding. Displaying in 8-bit color requires the Sun Ray server to convert all pixels to a pseudo 8-bit color stored in 24 bits of information before they are sent over the network. As a result, displaying in 8-bit color reduces the display quality and increases the server overhead, but does not reduce the bandwidth requirements.

5.1.3 Benchmarks

We ran the benchmarks described in Section 4 on each of the four thin-client platforms. We measured the platforms using both the standard unmodified benchmarks and their respective slow-motion versions. We used the network simulator to vary the network bandwidth between client and server to examine the impact of bandwidth limitations on thin-client performance. We measured performance at four network bandwidths, 128 Kbps, 1.5 Mbps, 10 Mbps,

Platform	Citrix MetaFrame (Citrix Win2K)	Terminal Services (RDP Win2K)	VNC Linux	Sun Ray
Display	1024x768, 8-bit	1024x768, 8-bit	1024x768, 8-bit	1024x768, 24-bit
Transport	TCP/IP	TCP/IP	TCP/IP	UDP/IP
Options	Disk cache off, memory cache on, compression on	Disk cache off, memory cache on, compression on	Hextile encoding, copyrect on	N/A

Table 2: Thin-client platform configurations.

and 100 Mbps, roughly corresponding to ISDN, DSL/T1, and LAN network environments, respectively.

To run the Web Text Page Load benchmark, we used Netscape Navigator 4.72, as it is available on all the platforms under study. The browser's memory cache and disk cache were cleared before each test run. In all cases, the Netscape browser window was 1024x768 in size, so the region being updated was the same on each system. Nevertheless, Netscape on Windows 2000 performs somewhat differently from Netscape on Linux and Solaris. For instance, in the Unix version, fonts appear smaller by default and a blank gray page appears between page downloads. These effects would tend to increase the amount of data that would need to be transferred on screen updates using a Unix-based thin-client platform. Our experience with various thin-client platforms indicate that these effects are minor in general, but should be taken into account when considering small thin-client performance differences across Unix and Windows systems.

To run the MPEG1 Video benchmark, we used Microsoft Windows Media Player version 6.4.09.1109 for the Windows-based thin clients and MpegTV version 1.1 for the Unix-based thin clients. In order to facilitate a fair comparison between all platforms despite using two different players, we configured the two players so they had the same size video window and otherwise appeared as similar as possible. Since the only portion of the display that is updated is the video window, both Unix- and Windows-based thin clients are effectively performing the same tasks.

5.2 Web Benchmark Results

5.2.1 Standard Benchmark Results

Figure 4 and Figure 5 show the results of running the unmodified Web Text Page Load benchmark on each of the thin-client platforms. Figure 4 shows the total latency for the unmodified benchmark on each platform. To provide some context for these results, a per-page latency of less than one second has been shown to be desirable to ensure that the flow of a user's browsing experience is not interrupted [20]. Given the 109 web pages in the Web Text Page Load benchmark, a total latency of less than 109 seconds is necessary for good performance.

At first glance, it appears that VNC performs extremely well, maintaining the same low latency across all bandwidths and outperforming the other platforms, 46% faster than its nearest competitor, RDP, at 100 Mbps, while Sun Ray appears to perform much worse than the other platforms, 20% slower than RDP at 100 Mbps. In addition, both Citrix and VNC still appear to

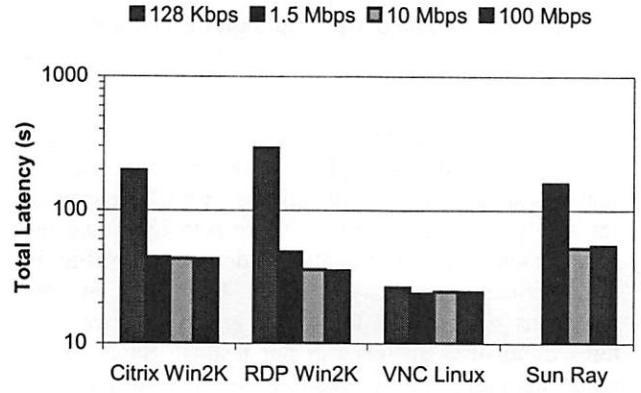


Figure 4: Total latency for unmodified web benchmark. Using Sun Ray, the benchmark did not complete at 128 Kbps.

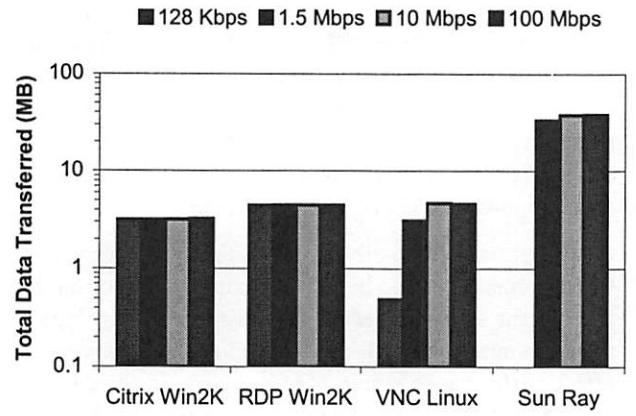


Figure 5: Total data transferred for unmodified web benchmark.

be performing well on the benchmark even at 128 Kbps with average per-page download speed of less than 1 second. However, examining the data transferred results in Figure 5 shows that VNC discards a substantial amount of display data at lower bandwidths, while the other platforms transmit a consistent amount of data and slow down playback as necessary.

This highlights the problems with the results from the standard benchmark. Because we do not know exactly how the data is being encoded and compressed under each platform, we have no way of establishing a baseline for how much data should be transferred to the client by each system. As a result, we have no way of knowing whether the pages are being fully transferred to the client, even at 100 Mbps. We also cannot be sure that each platform is transmitting updates corresponding to the same pages, so the data transfer results are not an accurate measure of the relative efficiency of the platforms. As a result, we cannot draw conclusions about the relative performance of the

systems when they are effectively being tested on different sequences of pages.

Visual observation of the platforms during the course of the test revealed another weakness of the standard benchmark. The pages stream by at such a fast rate that the sequence is not a realistic model of web browsing behavior. Real users typically do not interact with a browser in a rapid-fire manner but rather wait for a page to load before clicking on to the next page. This rapid rate causes a “pipelining” effect that hides the latency that results when each page is loaded from a standing start, which would be experienced in typical use.

5.2.2 Slow-Motion Benchmark Results

Figure 6 and Figure 7 show the results of running the slow-motion version of the Web Text Page Load benchmark on the four thin-client platforms. Figure 6 shows the total latency for downloading the 109 web pages, calculated as the sum of the individual page download latencies. A progressive improvement in performance with increased bandwidth is now visible for all of the platforms, even VNC Linux, which showed exaggerated performance at lower bandwidth under the unmodified benchmark.

As shown in Figure 7, the amount of data transferred now remains almost constant for all of the platforms across all bandwidths. However, we note that VNC transmits slightly less data at lower bandwidths because it uses a client-pull update policy in which each display update is sent in response to an explicit client request. At low network bandwidths, each display update takes longer to transmit, resulting in the client sending fewer update requests and receiving fewer display updates. The unsent interim updates are merged by the server. This does not affect the overall results as we are only interested in the total per-page latency for displaying the entire viewable web page. The absence of interim updates received at high bandwidths when the client can send more update requests does not affect the final visual quality or per page download latency.

Comparing Figure 4 and Figure 6, the measurements show that the total latency for the slow-motion benchmark is from 10% (for Sun Ray) to 63% (for VNC) higher than for the standard unmodified benchmark. There are three reasons for the difference in latency. First, none of the thin-client platforms discard display updates for the slow-motion benchmark. A comparison of Figure 5 and Figure 7 shows that VNC no longer discards display updates for pages in the slow-motion benchmark as it did for the unmodified benchmark. VNC transfers more data in the slow-motion case even at 100 Mbps, indicating that VNC

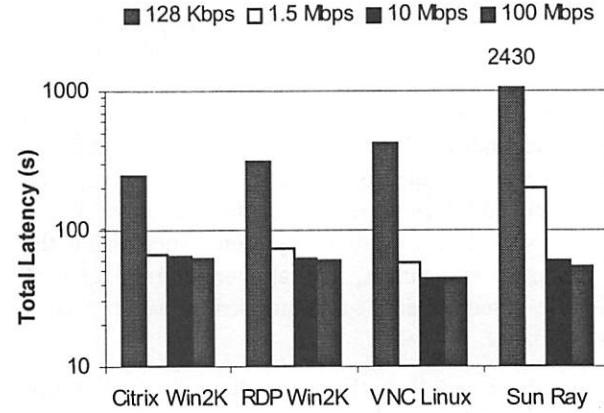


Figure 6: Total latency for slow-motion web benchmark.

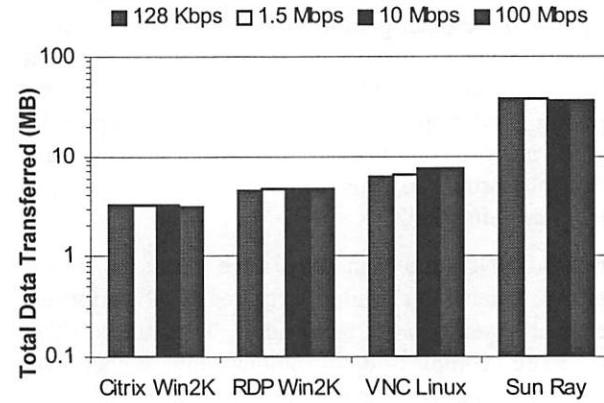


Figure 7: Total data transferred for slow-motion web benchmark.

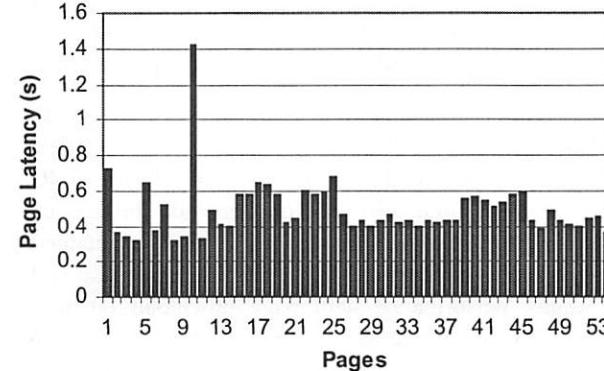


Figure 8: Per-page latency for VNC Linux running the slow-motion benchmark at 100 Mbps.

was discarding data even at the highest bandwidth when running the unmodified benchmark. Second, in using the slow-motion benchmark, each web page is downloaded from a standing start after the previous page is completely downloaded. None of the latency is hidden by “pipelining” page downloads. Third, for

Citrix and RDP, there were two web pages, pages 23 and 49 in the second iteration of downloading the pages, that consistently took 3-4 seconds to download for the slow-motion benchmark that did not take as long in the unmodified benchmark. We discovered that the long delays were due to an unusual interaction between Netscape and these two thin-client platforms. While these extra delays were not present when using the unmodified benchmark, the slow-motion benchmark provides a more realistic measurement of web browsing performance.

Figure 6 shows that all of the thin-client platforms deliver acceptable web browsing performance at LAN network bandwidths and that all of the platforms except Sun Ray provide sub-second performance at 1.5 Mbps as well. As shown in Figure 7, since Sun Ray provides higher quality 24-bit display as opposed to the 8-bit displays of the other platforms, it consumes much more network bandwidth, resulting in lower performance at low bandwidths. Note that none of the platforms provide good response time at 128 Kbps, despite the claims made by Citrix and Microsoft that their thin-client platforms can deliver good performance even at dialup modem speeds.

Overall, VNC and Sun Ray were faster at higher network bandwidths while Citrix and RDP performed better at lower network bandwidths. This suggests that the more complex optimizations and higher-level encoding primitives used by Citrix and RDP are beneficial at lower network bandwidths when reducing the amount of data transferred significantly reduces network latency. However, the simpler architectures of VNC and Sun Ray have lower processing overhead and hence perform better when bandwidth is more plentiful and data transfer speed is not the dominant factor.

Slow-motion benchmarking also allows us to obtain actual per-page results. Figure 8 shows a subset of the per-page latency results for one of the platforms, VNC. Due to space limitations, we only include the latency, but the per-page data transferred can also be obtained. For all pages except one, VNC provides excellent web browsing performance with page download latencies well below a second. Much information about the way the different platforms handle different types of pages is hidden by the aggregate results, but with the standard unmodified benchmark it is impossible to obtain the per-page data.

To further validate the accuracy and appropriateness of the slow-motion benchmarking technique, we internally instrumented the open-source platform VNC. By instrumenting VNC, we could obtain end-to-end latency measurements that also completely include any

client latency. We repeated the experiments with the instrumented version of VNC and compared the results with the packet capture data. The slow-motion results using network monitoring were verified to be within 4.3% of the instrumented VNC results in measuring the total data transferred and within 1.1% in recording the total latency. Furthermore, there was little variance in the results corresponding to each individual page across multiple runs. Of all the thin-client platforms measured, VNC had the highest client load and yet the slow-motion network monitoring results and internal instrumentation results showed little difference. The main reason for this is that the VNC client sends a message back to the server when it has finished processing the latest display update. As a result, the packet traces completely capture the client latency without direct client instrumentation.

An important benefit of slow-motion benchmarking for measuring interactive responsiveness is the reproducibility of the results. One way to measure interactive performance is to monitor actual user activity, but it is essentially impossible for a user to repeat the exact same set of experiments with the exact same timing characteristics. In contrast, slow-motion benchmarking can be used to provide better reproducibility of results. We gauged the reproducibility of the slow-motion benchmark data by calculating the standard deviation after five trials of each test. The largest standard deviation observed was 4.7% of the mean, but typically 3% or lower.

5.3 Video Benchmark Results

5.3.1 Standard Benchmark Results

Figure 9 and Figure 10 show the results of running the standard unmodified MPEG1 Video benchmark on the four thin-client platforms. Figure 9 shows the playback time for the MPEG video benchmark at the ideal frame rate of 24 fps. Unfortunately, playback time remained relatively static on all of the platforms and did not

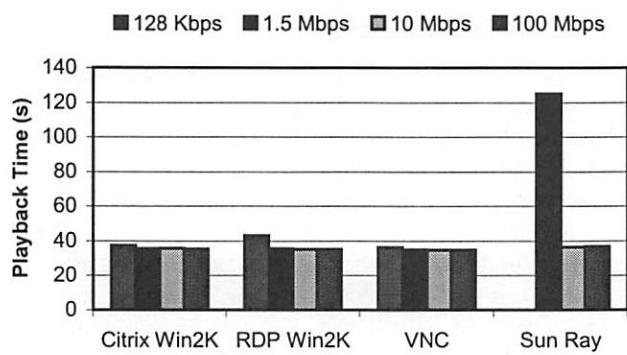


Figure 9: Playback time for unmodified video benchmark. Using Sun Ray, the benchmark did not complete at 128 Kbps.

correspond with the subjective performance, which degraded rapidly at lower bandwidths.

This subjective observation is supported by the data transfer measurements. Figure 10 shows the data transferred during playback, which degrades rapidly at lower bandwidths even when playback time remains low. For instance, the data transferred by VNC at 100 Mbps is 30 times greater than that transferred at 128 Kbps despite the near-constant playback time. Clearly, we cannot use the playback time alone as a measure of the video quality because not all the frames are being fully displayed. The amount of data transferred must be incorporated into any metric of video quality.

We could represent the video quality as a percentage of the ideal data transfer rate. However, this ideal data transfer rate cannot be determined with the unmodified benchmark. If we assumed that the 100 Mbps rate was the ideal, we might conclude that all of the platforms perform well at both 100 Mbps and 10 Mbps: they maintain a high playback time and transmit roughly the same amount of data at both bandwidths. This does not correlate with the subjective performance: visually, only Sun Ray achieved good performance even at 100 Mbps.

5.3.2 Slow-Motion Benchmark Results

Slow-motion benchmarking again allows us to clarify the picture. Figure 10 also shows the amount of data transferred when the benchmark was run in slow-motion at a frame rate of 1 fps with network bandwidth of 100 Mbps. At this frame rate, bandwidth limitations were not an issue and each frame of the video was transmitted separately and fully displayed on the client before the subsequent frame was begun. This yields a baseline by which to measure the results from the standard benchmark, using the formula for video quality described in Section 4.2.

Figure 11 shows this measure of video quality for each of the platforms. We can now obtain a clearer picture of how well each of the platforms perform at high bandwidths and in comparison to each other, despite the nearly-level playback time seen in Figure 9.

Out of all the thin-client platforms, Sun Ray alone achieves good performance, with 96% video quality at 100 Mbps despite the fact that it sends an order of magnitude more data than any other platform at 24 fps. None of the other platforms has good performance even at LAN bandwidths. The fact that Sun Ray sends much more data than any other platform indicates that the poor performance of these other platforms at 100 Mbps is not due to bandwidth limitations but is rather due to

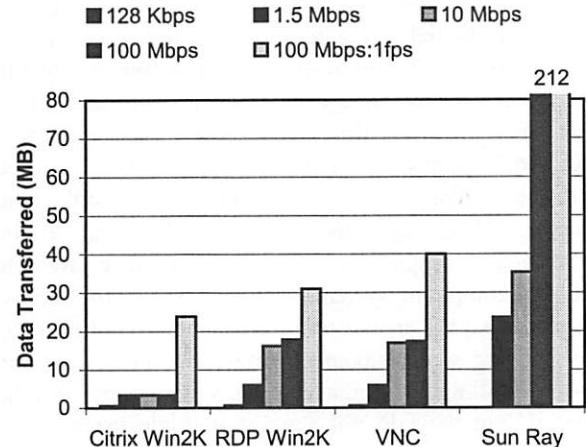


Figure 10: Total data transferred in unmodified video benchmark at 24 fps, and in the slow-motion video benchmark at 100 Mbps bandwidth and 1 fps. Sun Ray data transferred at 100 Mbps was equivalent at both frame rates.

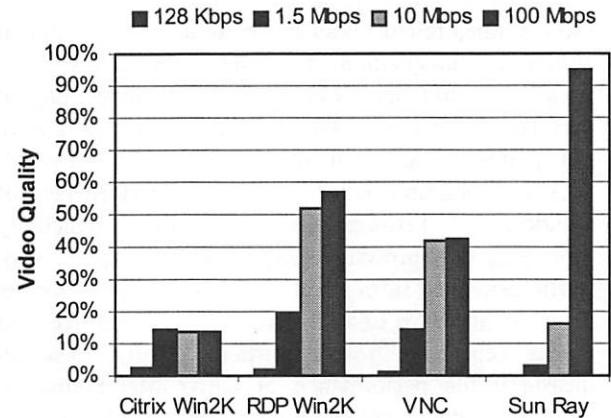


Figure 11: Video quality as percentage of data transferred in the slow-motion video benchmark.

their display update mechanisms, which are poorly suited to video applications.

6 Related Work

In this paper, we have focused on thin-client systems in which both applications and the window system are completely executed on the server. These systems are the most popular thin-client systems today and many of them have been developed [4, 5, 15, 16, 24, 26, 27, 30, 32].

Three other types of systems that are sometimes referred to as thin-client systems are network window systems, browser-based systems, and remote control computing systems. The most notable example of a network window system is the X Window system [25]. Unlike the systems discussed in this paper, X runs the

window system on the client and as a result requires more substantial client resources in order to perform well. To run X applications over lower bandwidth networks, a low-bandwidth X (LBX) proxy server extension [13] was developed and released as part of X11R6.3. Browser-based systems employ a web browser client as a user interface to an application server. These systems require applications to be modified to support a web-based interface. Remote control computing systems such as Laplink [14] and PC Anywhere [21] enable users to remotely control a PC by sending screen updates to remote client PCs. They also run all application and window system logic on the server, but they do not support multiple users at the same time.

There have been several studies of thin-client performance that have focused on evaluating one or two systems. Danskin conducted an early study of the X protocol [7] and Schmidt, Lam, and Northcutt examined the performance of Sun Ray [26]. Both of these studies relied on source code access for internal system instrumentation. Thin-client platform vendors such as Citrix and Microsoft have done internal performance testing of their products as well, but have not published any reliable experimental results [4, 16,17]. Wong and Seltzer studied the performance of Windows NT Terminal Server for office productivity tools and web browsing [33] by monitoring network traffic generated from a real user session. This provides a human measure of user-perceived performance, but makes repeatable results difficult. Tolly Research measured the performance of Citrix MetaFrame on various scripted application workloads [31], however the study suggests that problems in using standard scripted application workloads as described in this paper were not properly considered.

A few performance studies have compared a wider range of thin-client systems. Some of our previous work led to the development of slow-motion benchmarking [19]. Howard has presented performance results for various hardware thin-clients based on tests from the i-Bench benchmark suite [12]. This work suffers from the same problems in measurement technique that we described in Section 2. It relies on the results reported by the standard benchmarks, which only measure benchmark performance at the server-side. In addition, the work was based on Microsoft Internet Explorer 5.01, which does not properly interpret the Javascript onLoad handler used in the i-Bench Web Text Page Load benchmark. This causes successive pages to begin loading before the previous pages have fully displayed, resulting in unpredictable measurements of total web page download latencies.

Netscape Navigator 4.7 does not suffer from this problem, which is one of the reasons we used this browser platform for our work.

7 Conclusions and Future Work

We have introduced *slow-motion benchmarking*, a new measurement technique that requires no invasive instrumentation and yet provides accurate measurements for evaluating thin-client systems. Slow-motion benchmarking introduces delays into standard application benchmarks to isolate the visual components of those benchmarks. This ensures that the components are displayed correctly on the client when the benchmark is run, even when the client display is decoupled from the server processing as in many thin-client systems. Slow-motion benchmarking utilizes network traffic monitoring at the client rather than relying on application measurements at the server to provide a more complete measure of user-perceived performance at the client.

We have demonstrated the effectiveness of slow-motion benchmarking on a wide range of popular thin-client platforms. Our quantitative results show that slow-motion benchmarking provides far more accurate measurements than standard benchmarking approaches that have been used for evaluating thin-client systems. Our comparisons across different thin-client systems indicate that these systems have widely different performance on web and video applications. Our results suggest that current remote display mechanisms used in thin-client systems may be useful for web browsing at lower network bandwidths. However, these same mechanisms may adversely impact the ability of thin-client systems to support multimedia applications.

We are currently using slow-motion benchmarking to evaluate a wide range of thin-client platforms in different network environments. As ASPs continue to increase in popularity, one important area of research is evaluating the performance of thin-client computing in wide-area network environments. Slow-motion benchmarking provides a useful tool for characterizing and analyzing the design choices in thin-client systems to determine what mechanisms are best suited for supporting future wide-area computing services.

8 Acknowledgements

We thank Haoqiang Zheng for developing the instrumented version of VNC used in our experiments. Haoqiang, Albert Lai, Rahul Joshi, and Carla Goldburg, all assisted with many of the thin-client performance measurements and helped set up the thin-client testbed.

Allyn Vogel of Ziff-Davis Media, Inc. provided us with valuable information on i-Bench. We also thank the anonymous USENIX referees and our shepherd Vern Paxson, who provided helpful comments on earlier drafts of this paper. This work was supported in part by an NSF CAREER Award and Sun Microsystems.

References

1. AG Group, Inc., Etherpeak 4, <http://www.aggroup.com>.
2. Boca Research, "Citrix ICA Technology Brief," Technical White Paper, Boca Raton, FL, 1999.
3. M. Chapman, <http://www.rdesktop.org>.
4. Citrix Systems, "Citrix MetaFrame 1.8 Backgrounder," Citrix White Paper, June 1998.
5. B. C. Cumberland, G. Carius, A. Muir, *Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference*, Microsoft Press, Redmond, WA, Aug. 1999.
6. J. Curtis, "Port Mirroring: The Duplex Paradox," *Network World Fusion*, Oct. 1998.
7. J. Danskin, P. Hanrahan, "Profiling the X Protocol," *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Nashville, TN, 1994.
8. Desktop.com, <http://www.desktop.com>.
9. Expertcity.com, "Desktop Streaming Technology and Security," Expertcity White Paper, Santa Barbara, CA, 2000.
10. Futurelink, <http://www.futurelink.net>.
11. GraphOn GO-Global, <http://www.graphon.com>.
12. B. Howard, "Thin Is Back," *PC Magazine* 19(7), Ziff-Davis Media, New York, NY, Apr. 2000.
13. "Broadway / X Web FAQ," <http://www.broadwayinfo.com/bwfaq.htm>.
14. LapLink.com, Inc., *LapLink 2000 User's Guide*, Bothell, WA, 1999.
15. T. W. Mathers, S. P. Genoway, *Windows NT Thin Client Solutions: Implementing Terminal Server and Citrix MetaFrame*, Macmillan Technical Publishing, Indianapolis, IN, Nov. 1998.
16. Microsoft Corporation, "Microsoft Windows NT Server 4.0, Terminal Server Edition: An Architectural Overview," Technical White Paper, Redmond, WA, 1998.
17. Microsoft Corporation, "Windows 2000 Terminal Services Capacity Planning," Technical White Paper, Redmond, WA, 2000.
18. J. Nieh, S. J. Yang, "Measuring the Multimedia Performance of Server-Based Computing," *Proceedings of the 10th International Workshop on Network and Operating System Support for Digital Audio and Video*, Chapel Hill, NC, June 2000.
19. J. Nieh, S. J. Yang, and N. Novik, "A Comparison of Thin-Client Computing Architectures," Technical Report CUUCS-022-00, Department of Computer Science, Columbia University, Nov. 2000.
20. J. Nielsen, *Usability Engineering*, Morgan Kaufman, San Francisco, CA, 1994.
21. PC Anywhere, <http://www.symantec.com/pcanywhere>.
22. Personable.com, <http://www.personable.com>.
23. T. Richardson, Q. Stafford-Fraser, K. R. Wood and A. Hopper, "Virtual Network Computing," *IEEE Internet Computing*, 2(1), Jan./Feb. 1998.
24. The Santa Cruz Operation, "Tarantella Web-Enabling Software: The Adaptive Internet Protocol," A SCO Technical White Paper, Dec. 1998.
25. R. W. Scheifler and J. Gettys, "The X Window System," *ACM Transactions on Graphics*, 5(2), Apr. 1986.
26. B. K. Schmidt, M. S. Lam, J. D. Northcutt, "The Interactive Performance of SLIM: A Stateless, Thin-Client Architecture," *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, Dec. 1999.
27. A. Shaw, K. R. Burgess, J. M. Pullan, P. C. Cartwright, "Method of Displaying an Application on a Variety of Client Devices in a Client/Server Network," US Patent US6104392, Aug. 2000.
28. B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 2nd ed., Addison-Wesley, Reading, MA, 1992.
29. Shunra Software, *The Cloud*, <http://www.shunra.com>.
30. Sun Microsystems, *Sun Ray 1 Enterprise Appliance*, <http://www.sun.com/products/sunray1>.
31. Tolly Research, "Thin-Client Networking: Bandwidth Consumption Using Citrix ICA," *IT clarity*, Feb. 2000.
32. Virtual Network Computing, <http://www.uk.research.att.com/vnc>.
33. A. Y. Wong, M. Seltzer, "Evaluating Windows NT Terminal Server Performance," *Proceedings of the Third USENIX Windows NT Symposium*, Seattle, WA, July 1999.
34. S. J. Yang, J. Nieh, "Thin Is In," *PC Magazine*, 19(13), Ziff Davis Media, NY, July 2000.
35. Ziff-Davis, Inc., i-Bench version 1.5, <http://i-bench.zdnet.com>.

An Architecture for Secure Generation and Verification of Electronic Coupons

Rahul Garg

Parul Mittal

Vikas Agarwal

Natwar Modani

{grahul, mparul, avikas, mnatwar}@in.ibm.com

IBM India Research Lab.

Abstract

Coupons are very useful mechanism for carrying out different marketing management functions like sales promotion, brand promotion, and inventory management. With the advent of Internet shopping and online stores, there is an immediate need for an electronic equivalent of traditional paper coupons. Security issues such as coupon tampering, exchange, duplication and double spending become very important for electronic coupons. Although the security issues in electronic coupons appear to be similar to those in electronic cash systems, there are significant differences that require the design of a different protocol to carry out secure coupon transactions.

In this paper we describe a system for secure generation and verification of electronic manufacturer and store coupons. The proposed solution is based on a third party centralized coupon mint which carries out the check for double spending, similar to online electronic cash systems. However, unlike electronic cash systems, the coupon mint remains completely unaware of the promotion details (the amount of discount, product details etc.) and simply provides an infrastructure for online coupon verification. Thus, the coupon mint service can be provided by semi-trusted third parties different from manufacturers. The proposed system is inherently distributed and scalable. It lets different manufacturers independently choose their own promotion and targeting policies (without involving the coupon mint) and the coupon mint service provider.

The system also offers several new types of coupons like aging coupons, growing coupons, random value coupons, and early bird coupons which were not practical by using traditional paper coupons (and not possible by using the electronic cash protocols).

1 Introduction

Coupons have been traditionally used to carry out different marketing management functions like sales promotion, brand promotion, and inventory management [1]. As a result of advancement in the Internet and e-commerce technologies, a large number of online stores offering a variety of products and services have been opened. These electronic stores also like to issue electronic coupons to their customers for sales promotion and other marketing management functions.

There can be different types of coupons depending on the issuing authority (manufacturer or store) [2], whether they are targeted for a set of potential customers or they are untargeted, whether their distribution is limited or not, and whether the value of the offer is fixed and known in advance to the customer. Promotions are designed using these coupon types to achieve a specific marketing objective. For instance, a manufacturer may issue some targeted coupons for limited distribution for product trials and a store may issue coupons for unlimited distribution to get rid of unsold inventory of perishable items.

The online nature of electronic coupons makes them prone to various kinds of frauds. A number of manufacturers are not adopting Internet coupons because of security concerns [3]. As a result, the security issues [3] have become even more important for electronic coupons. There is an immediate need for a generic electronic coupon system that is secure, can be trusted and can offer a wide variety of coupon types including those which are currently in practice.

Double spending is the most difficult security problem to handle in case of manufacturer electronic

coupons. Several solutions to this problem have been proposed in the context of electronic cash [4, 5, 6]. However, none of the proposed solutions are applicable for electronic coupons because of the differences in the way electronic cash and electronic coupons are used. A bank may be willing to provide a highly available online verification infrastructure for its business, whereas a manufacturer may not be willing to provide the same for its electronic coupons. If double spending is detected at a later stage, banks may be able to recover the doubly spent money from the customer responsible for it (through debit or litigation), whereas a manufacturer may not be able (and willing) to do so for the doubly redeemed electronic coupons. Customers may be willing to carry a special card for their electronic cash, but not for electronic coupons.

Currently, there are a number of Internet web sites that offer coupons online, as an image or a bar code, that the user can print on a local printer and use in a particular physical store. Such coupons are not really electronic coupons as they can only be used at a physical store. They are best described as traditional paper coupons distributed on the Internet.

Online stores on the Internet have also started issuing coupons. Most of the current offerings give coupons that can only be used at the issuing store and are very limited. Kumar et al. [2] have proposed an electronic coupon architecture which is limited to single store electronic coupons. Currently, there is no secure system for generic electronic manufacturer coupons.

Other related problems are that of “hit-shaving” [7] and “hit-inflation” [8]. Hit shaving occurs when a target web site (advertiser) omits the references of some of the hits to its web site. Hit inflation occurs when a referrer’s site (say a portal) artificially generates hits to the advertiser’s site that do not correspond to a genuine user’s visit to the site. These problems have become important especially in the context of advertisement on the Internet that use a click-through payment program. A detailed study on these can be found in Reiter et al. [7] and Anupam et al. [8].

In this paper we describe an architecture for secure generation and verification of electronic coupons of different types. The proposed solution is based on a third party centralized coupon mint which carries out the check for double spending. However, unlike electronic cash systems, the coupon mint remains

completely unaware of the promotion details (the amount of discount, product details etc.) and simply provides an infrastructure for online coupon verification. The system also offers several new types of coupons like aging coupons, growing coupons, random value coupons, and early bird coupons, which were not possible by using traditional paper coupons (and the electronic cash protocols).

We begin by describing the different types of paper coupons that are currently in practice in Section 2. We describe newer types of electronic coupons including aging coupons, growing coupons, random value coupons, early bird coupons that can be provided by our architecture. We also discuss the important security issues in a generic electronic coupon system and describe why the solutions proposed for electronic cash systems are not directly usable in an electronic coupon system. We describe our architecture and protocol in detail in Section 3. We also discuss how different types of frauds can be prevented by the proposed architecture. We discuss some implementation issues in Section 4 and we describe our implementation in brief in Section 5. We present the performance of our prototype implementation in Section 6 and conclude in Section 7.

2 Electronic Coupons: Possibilities and Pitfalls

There are several types of coupons. In this section, we first introduce the different types of coupons and then discuss security issues for a generic online electronic coupon system. We also discuss why protocols proposed for electronic cash systems cannot be used in their present form in an electronic coupon system.

2.1 Coupon Classification

Manufacturer vs. Store coupons. This classification is based on the coupon issuing authority. A store may distribute discount coupons on a few products to attract customers to the store. These coupons can only be redeemed at the particular store. The cost of such a promotion is borne completely by the store issuing the coupons. After collecting some relevant information (needed for future profiling, evaluating the scheme etc.), the

redeemed coupons may be discarded by the store. Such coupons are called store coupons. On the other hand, a leading brand manufacturer may periodically issue discount coupons to its loyal customers. The customer may take these coupons to any store for redemption. Such coupons are called manufacturer coupons. The cost of such a promotion is borne entirely by the manufacturer. The stores claim the discount given to the customer by sending all the redeemed coupons to their respective manufacturers. This process is called coupon clearing. The manufacturer uses the cleared coupons to gather important sales and redemption information. In addition to the discount amount, the manufacturer may also pay a handling fee to the stores for each manufacturer coupon redeemed at these. There may be coupons that fall between manufacturer coupons and store coupons, such as manufacturer coupons that can be redeemed at selected stores, or coupons that can be redeemed on any store of a retail chain, or coupons valid only at stores participating in a given program.

Targeted vs. Untargeted coupons. Many times manufacturers or stores want to issue certain coupons to only a selected group of customers. For instance, a leading brand may wish to issue coupons targeted to the regular customers of a competitor brand, to induce brand switching. A targeted coupon is intended for a particular customer (or a set of customers), whereas an untargeted coupon may be used by anyone.

Limited distribution vs. Unlimited distribution coupons. The coupon issuers often want to control the number of coupons of a particular type that are distributed. By limiting the distribution, the issuer can limit the amount of discount to be given, and hence estimate the overall cost of the promotion (say in a product trial promotion). Such coupons are called limited distribution coupons. Sometimes manufacturers or stores prefer to distribute a large number of discount coupons instead of a price mark-down. Such coupons are called unlimited distribution coupons.

Variable value coupons. Finally, yet another classification of coupons is based on the coupon value. Most of the printed coupons have fixed value known in advance. However, with the proposed electronic coupon system, it is possible that the coupon value is determined dynamically, based on certain parameters. This results in a number of exciting possibilities, such as gaming, lottery etc., which may

be used for defining more effective promotions. An *early bird coupon* is one where the first “ k ” customers who bring the coupon to a store get the discount. An *aging coupon* is one whose value decays with time. A *growing coupon* is one whose value increases with time. A *random value coupon* is the one whose value is not known at the time of issuing. The value of such a coupon is known only after a purchase is made. However, the range and the probability distribution of the possible values may be known in advance. A lottery ticket is an extreme example of a random value coupon, where the coupon value is either zero or a large sum (the lottery prize) and the statistical distribution of possible coupon values is known in advance. The coupon value is known only after the purchase (of lottery ticket) is made.

2.2 Security Issues

Tampering and double spending are two important security issues in an online currency of any form. In the context of electronic coupons, a customer may change the discount amount or other terms (such as validity period) of an electronic coupon to get an illegitimate discount. In the case of manufacturer coupons, even the retailers may alter the coupon before sending it for clearing. The problem of tampering is usually solved by using digital signatures [9, 10].

The problem of double spending becomes severe especially in the case of manufacturer coupons. Since electronic coupons can be duplicated easily, a customer may use the same manufacturer coupon at two different retailers. The manufacturer will not know this until both retailers send their coupons for clearing. In a more contrived scenario, a group of retailers may duplicate and share the manufacturer coupons redeemed at their stores and blame customers for double spending.

There are online as well as off-line solutions for the double spending problem in the electronic cash literature. The off-line solutions that prevent double spending require the users to store the cash in a special *tamper-proof* [11] hardware. Such a solution may be difficult to use for electronic coupons as users may not be willing to get a special hardware just for storing their electronic coupons. There is another class of off-line solutions [4, 5, 6] that do not prevent double spending but detect the identity of user involved in such a fraud. Double spending is

detected only when two or more copies of the money spent twice is deposited back into the bank. While such off-line solutions may work for electronic cash, they may not work for electronic coupons because of the nature of relationship between customers and the manufacturers. Customers do not have accounts with manufacturers and the manufacturers may not be able to charge (or sue) customers even if double spending is detected at a later stage.

The online solutions proposed for preventing double spending are similar to a system where the coupon clearing is done online at the time of purchase. In this case, if a coupon is redeemed for the second time, the manufacturer would inform the retailer and the retailer would refuse to accept the coupon. To implement an online coupon clearing system, the manufacturer will need to implement a highly reliable and available coupon clearing infrastructure, which is likely to fall beyond the abilities and interests of most manufacturers. The manufacturers may not be willing to completely trust third parties to provide this service to them, especially because it implies giving them access to their valued customer and sales data. Therefore, an approach is needed where retailers can verify the validity of coupons without requiring any online support from manufacturers.

Another important issue for electronic coupons is to prevent coupon trading among customers (unlike electronic cash systems where the goal is the facilitate the cash exchange among users). The purpose of targeting is defeated if the customers are able to freely exchange or transfer their coupons, possibly in return for money or other coupons. There are already several Internet web sites for coupon trading. In addition, the manufacturers would like to track the customers and their coupon usage patterns (unlike electronic cash systems where banks will like to provide anonymity to their customers) for evaluating the success of a promotion and for future targeting and profiling.

The manufacturer should be able to relate the coupons redeemed by a customer at a retailer to the sales of intended products. If this is not done, then the retailers may just collect the coupons and send them to manufacturers for clearing. The manufacturers should also be able to check if the retailers are properly checking all the terms and conditions of the coupons.

Finally, in the case of variable value coupons, the

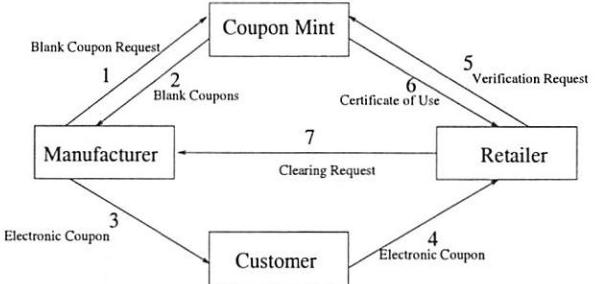


Figure 1: Components in the Electronic Coupon System.

coupon value is not fixed and depends on time at which the coupon is being redeemed, the number of coupons of that promotion which have been redeemed earlier, and a random element. In this case, the retailer should be able to determine the correct coupon value without any online support from the manufacturer. In addition, the customer should be able to check that the coupon value determined by the retailer is indeed according to the parameters specified in the coupon. Finally, the manufacturer should be able to check that the coupon value reported in the coupon sent for clearing is indeed the correct coupon value.

In the following section, we describe an architecture for a coupon system which addresses all the security issues discussed above. The architecture is based on a third party *semi-trusted* centralized *coupon mint* which checks for double spending and helps in determining the fair coupon value, without knowing any other details about the coupon.

3 System Description

Figure 1 shows the main entities in a generic electronic coupon system. It consists of a coupon mint, a manufacturer, a customer and a retailer. A coupon mint is an independent third party service provider, which provides highly available online electronic coupon verification service to retailers on behalf of multiple manufacturers. The manufacturer represents the authority issuing coupons to promote its products or services. The customer represents the users to whom the coupons are issued and who make purchases of products or services from the retailers. The retailer represents an online store selling products and/or services.

```

<blankCouponRequest> ::= <numCoupons> [<classId>] [<validity>] [<manufId>]
<blankCoupon>      ::= <couponId> <blankCouponExpiry> [<timeCreation>] [<classId>]
                      [<manufId>] [<verificationUrl>] <signature>
<electronicCoupon> ::= <blankCoupon> <issuingAuthority> <discount> [<conditions>]
                      [<visual>] <signature>
<conditions>       ::= [<validityPeriod>] [<purchaseConditions>]
                      [<personalizationConditions>] [<otherConditions>]
<verificationRequest> ::= <blankCoupon> <invoiceNo> [<retailerInfo>]
<certificateOfUse>  ::= <blankCoupon> <invoiceNo> [<retailerInfo>] <numUsed>
                      [<numClassUsed>] <rand> <dateTime> <signature>
<clearingRequest>   ::= <electronicCoupon> <certificateOfUse>

```

Figure 2: Messages exchanged in the electronic coupon system. The terms enclosed in [] are optional.

Figure 2 shows the details of messages exchanged between these entities during the life cycle of an electronic coupon. All the messages are encrypted to ensure privacy.

In the following subsections we explain our electronic coupon protocol and the steps taken by each of the entities involved during coupon issue, redemption and clearing to safeguard against potential fraud.

3.1 Coupon Issuing

When a manufacturer needs to issue a coupon, it sends a *blank coupon request* to the coupon mint which then issues unique unforgeable blank coupons to the manufacturer. Unforgeability implies that when these coupons are presented to the coupon mint at a later stage: (a) it can recognize that these coupons have been generated by the coupon mint and (b) no other entity can produce a blank coupon which, with significant probability, may be recognized by the coupon mint as valid. Unforgeable coupons can be produced by using a long coupon identifier x and applying a keyed hash function [12] $f_k(x)$ to it, where k is the key known only to the coupon mint. The pair $(x, f_k(x))$ forms an unforgeable blank coupon. We use digital signatures, which are a special case of keyed hash functions, to generate unforgeable coupons as seen from the format of *blankCoupon* message in Figure 2.

The manufacturer then writes the coupon details on a blank coupon and digitally signs it to make it an authentic electronic coupon. Coupon details contain the discount amount, purchase conditions, va-

lidity period, personalization condition, other conditions and a human-readable description of the offer (such as an image or HTML text).

In the simplest case, the discount amount is simply a number. However, for variable value coupons, the discount amount is represented as a function which takes several parameters as input to compute the exact discount amount.

Validity period is represented as a start and end timestamp. The coupon is valid only in this period. In addition, a coupon may contain conditions indicating its validity during specified intervals of a day (lunch coupons), on specified days of a week (Sunday coupons), and in specified weeks of a month.

Purchase conditions represent the set of purchases required by a customer to obtain the discount. These may be a list of products (with given minimum units of necessary purchases), or products from a product family, or purchase totaling more than a given amount. For example, a *buy one and get one free* coupon will require the purchase of two units of a product and offer 100% discount on one unit of the product.

Personalization conditions ensure that only the targeted customers are able to redeem coupons. The simplest method to identify a customer is by a credit card number. Credit card based personalization conditions contain a one way hash function [12] of the targeted customer's credit card number, which is verified against the credit card number given at the time of online purchase. A one-way hash function ensures that the credit card number is not misused. These coupons may impose the restriction that the customer should use the same credit card for pur-

chase with which the coupon is personalized. Such coupons are difficult to issue and distribute as the customer's credit card number (or its one-way hash) may not be known at the time of coupon distribution.

Other personalization conditions may be based on customer's email address, IP address, Internet service provider (ISP), profile based on a cookie, membership number, customer name, address and zip code. It is easier for online sites to know a customer's name or email address. Therefore, personalization based on name and email address is easier to carry out than credit card based personalization. However it may be relatively difficult to verify. In this case the retailer may require the customer to register at its site (and supply personal details such as name, email address etc.) before the customer can redeem coupons. It is also possible to carry out reasonable targeting using a combination of the above personalization conditions.

A coupon may contain some other conditions for its validity. Some coupons may be valid at selected retailers, some may be valid only if a given payment method is used.

3.2 Redemption

When the customer presents a coupon to a retailer the retailer first checks the integrity of coupon by verifying the manufacturer's digital signature. The retailer knows the list of products it sells and hence the list of manufacturers which may potentially issue coupons. Therefore it does not need a public key infrastructure [13] for obtaining public keys. It may periodically get public keys from its manufacturers.

Secondly, the retailer checks if the purchase conditions, validity period, personalization condition, and other conditions mentioned in the coupon are valid at the time of redemption.

Finally, if all the conditions are met, the retailer checks if the coupon has been redeemed earlier by the customer by sending a *verification request* to the coupon mint. The verification request consists of the blank coupon part contained in the electronic coupon and an invoice number identifying the current sale transaction. The coupon mint responds to verification request with a digitally signed *certificate of use* which indicates the number of times (nu-

mUsed) a verification request for the given coupon has been sent by the retailers (and hence the number of times the coupon has been redeemed earlier). The retailer uses this information to check for double spending. Note that since all the messages are encrypted, only a manufacturer or a customer can get access to the blank coupon part of her coupons. Since blank coupons are unforgeable, no entity other than the customer herself can send fake verification requests to invalidate other customer's coupons.

For early bird coupons, the retailer needs to know the number of coupons of the same type redeemed so far. For this, each early bird coupon of a promotion scheme is tagged with a class identifier (classId), which identifies the promotion. The coupon mint keeps track of the number of coupons of each class redeemed so far and reports this number in the certificate of use (numClassUsed). The early bird coupons also contain an early bird condition which specifies the maximum permissible value of numClassUsed.

Variable value coupons are implemented by specifying a discount function in place of the discount amount. This function has four input parameters: date and time-stamp (dateTime), number of times the coupon has been redeemed earlier (numUsed), number of coupons of this coupon class redeemed so far (numClassUsed) and a random number (rand). The certificate of use contains all the input parameters of this discount value function. The retailer applies this function to the certificate of use and calculates the discount amount. Neither the retailer nor the customer has control over certificate of use. So neither can manipulate the discount amount to their advantage. The coupon mint has no incentive to give an incorrect certificate of use. The only possibility of fraud is when the retailer and the coupon mint (and optionally the customer) collude and design a certificate of use which computes the discount amount to their advantage.

A number of functions may be designed by the manufacturer for early bird, aging, growing, and random coupons. For example, for aging and growing coupons the function value depends on dateTime, for early bird coupons the function value depends on numClassUsed and for random value coupons the function value depends on rand. A number of interesting combinations of these coupon types are also possible.

3.3 Clearing

At a later stage, the retailer sends all the redeemed coupons along-with their certificate of use to their respective manufacturers for clearing. The manufacturer checks for integrity of electronic coupons and certificate of use by verifying the digital signatures. The certificate of use acts as a proof for the retailer that the coupon was redeemed at its store. For variable value coupons, the manufacturer checks using the function specified in the coupon, and its certificate of use, that the discount value was determined fairly by the retailer. If all these conditions are met, the manufacturer sends the required amount of money to the retailer.

The certificate of use also contains the retailer invoice number and optional sales related information (`retailerInfo`), which is used by the manufacturer to correlate a coupon redemption with a product purchase at the retailer. The manufacturer may also carry out periodic audits at retailers to check if the retailers are properly verifying all the conditions mentioned in the coupon.

It is to be noted that in the end, the manufacturers also get all the information about the usage pattern of redeemed coupons which can be used for designing future promotions. Also in the entire process, the coupon details are never revealed to the coupon mint. The manufacturers need to trust the coupon mint only for variable value coupons. If the coupon mint issues a valid certificate of use for a coupon that has been redeemed earlier, the manufacturer can detect it as it will finally get two valid certificate of use from two different retailers.

4 Discussion

The system described in Section 3 is a generic and secure system for electronic coupon generation and verification. It solves the problem of coupon forging, tampering, double spending, unwanted coupon exchanges, fair determination of coupon value for variable value coupons and ties coupon redemption to product purchase.

Separation of the online coupon verification functionality from coupon issue and distribution is the main contribution of this architecture. Online

coupon verification requires highly available online infrastructure whereas designing a promotion scheme, targeting, and coupon distribution requires data mining and domain-specific marketing knowledge. By separating the coupon verification and distribution, different manufacturers and stores can use their own promotion and coupon distribution policies, without maintaining online verification infrastructure of their own.

The architecture is generic enough to support manufacturer, store and group-of-store coupons for online stores. In the case of store coupons, the functionalities of manufacturer and retailer are co-located at the store. For group-of-stores coupons, the “other conditions” in the coupon contain a list of member stores where the coupon may be redeemed.

All verification requests coming to a centralized server raises serious scalability concerns. However it is easy to see that there may be several such coupon mints providing the verification service. Thus a manufacturer has a choice of selecting a coupon verification provider. It may also dynamically switch to a new verification provider, in case it finds the service of one unacceptable. The manufacturer writes a verification URL in the coupon which the retailer follows in order to send verification request. The coupon mint may also put a verification URL in the blank coupon to distribute its verification load across multiple verification servers.

The coupon mint needs to store information about the coupons redeemed only for a limited period. Each blank coupon has an expiry date. The validity period of a coupon must entirely be contained within this expiry date. The coupon mint keeps a record for each redeemed coupon only till its blank coupon expiry. Thus, if a verification request for an already redeemed coupon comes after its blank coupon expiry, the coupon mint simply rejects the request on the basis of expiry.

5 Implementation

The system software comprises of three main components, one each for the coupon mint, the manufacturer and the retailer. Optionally a fourth component, for storage of customer’s electronic coupons, is also provided. Each component is a stand alone Web based entity implemented as a Java servlet.

The cryptographic operations are written in the C language using Java Native Interface (JNI) for performance reasons. In order to adhere to standards, we used XML [14] to encode all the messages and SSL (TLS) [15]¹ to encrypt them.

5.1 Coupon Mint Component

The coupon mint component receives blank coupon requests from the manufacturers and verification requests from the retailers as HTTP/POST [16] messages and sends blank coupons or certificates of use as an XML document in responses.

5.2 Manufacturer Component

The manufacturer component allows multiple manufacturers to define different promotions, through a Web based interface. For each promotion, the manufacturer supplies the promotion details, such as the number of coupons to be distributed, the discount amount, the product ID etc. The software obtains an appropriate number of blank coupons from the coupon mint, writes promotion specific information on them and stores them.

The manufacturer component also provides an HTTP interface to distribute the coupons. This interface may be used to flash electronic coupons as banner advertisements on various Internet web sites. The manufacturer component gets a message to serve an electronic coupon advertisement. With the message, it gets the customer profile and personalization information. It shows a coupon image targeted to the customer. Clicking on a coupon image generates an HTTP/GET message at the manufacturer. The manufacturer component then writes the personalization information on the coupon, digitally signs it and, depending on the interface, either saves it on the customer's hard disk or uploads it to a third party *coupon storage service provider*.

5.3 Customer Component

A customer typically runs a web browser on a desktop to carry out on-line purchases. Any additional

¹TLS is the IETF proposed standard for SSL. The SSL protocol version 3.0 is available as an Internet draft.

software on the customer machine may discourage the customer from using electronic coupons. Besides, any additional customer side software also restricts the customer to use the same machine every time she wants to use the electronic coupons. Therefore we avoided any additional software on the customer side. Since the customer uses a web browser, customer-retailer interface and the customer-manufacturer interface is based on the HTTP protocol.

5.4 Retailer Component

The retailer component runs at various online stores. This component has to be integrated with the e-commerce software at each online store. The integration is done with the order processing module of the online store's e-commerce server. It is expected that use of Java to code this component would facilitate integration with different e-commerce servers.

Once a customer decides to buy some items from an online store and proceeds to process the order, the customer is given the option to redeem coupons. The retailer component provides an interface to upload electronic coupons from a customer's desktop or coupon storage provider, checks the coupons for applicability to the current purchase order, verifies the applicable coupons as described in Section 3, and finally reports the discount amount to the commerce server.

The uploading of coupons from the customer's desktop uses the "File Upload" proposed enhancement to HTML [17]. The interface with the coupon storage service provider comprises of a HTTP/POST request, sent to the coupon storage service provider to get the applicable coupons. The request contains the information about the items being purchased. The response from the coupon storage service provider contains the customer's coupons in XML, whose purchase conditions are satisfied by the items being purchased. However, such an interface may require a significant standardization effort. In general, there may be several purchase conditions in a coupon making some of the coupons mutually exclusive. There can be scenarios when there are multiple mutually exclusive coupons that can be used for the set of purchased items. The retailer component checks such situations and asks the customer to select a subset of non-mutually exclusive coupons.

An advanced decision support system to help the customer choose the best coupon subset, for a given purchase order, may be implemented in future.

In order to check the validity of selected coupon subset, the retailer component checks the manufacturer's signature on the coupons using manufacturer's public key. It re-checks the purchase conditions and checks the validity, personalization and other conditions. For this, the retailer component needs purchase order details and customer specific information (such as customer name, membership number, credit card number etc.) from the e-commerce server. Interfaces based on the purchase order identifier are defined, using which the retailer component can get customer-specific information from the e-commerce server for a given order. It is assumed that every purchase order in an e-commerce server will have a unique invoice number which is passed in the HTTP/POST messages to the retailer component.

Finally, for checking double spending and obtaining the certificate of use, the retailer component sends a HTTP/POST request containing the blank coupon part of the coupon, invoice number, and optional information. The coupon mint returns a certificate of use in XML format. The retailer component stores the coupon and the corresponding certificate of use for clearing with the manufacturer. The retailer component then computes the discount amount and returns it to the e-commerce server using a predefined API.

6 Performance Measurements

We measured the performance of our prototype implementation to estimate the resources required for a production system.

6.1 Experimental Setup

The testbed consists of one server and one client system. The server system is a 600 MHz Pentium-III with 128MB of RAM, and a 100 Mbps Ethernet network interface running RedHat Linux 6.2. The client system is a 233MHz Pentium-II with 96MB of RAM, running RedHat Linux 6.0. The client communicates with the server through a 100 Mbps LAN.

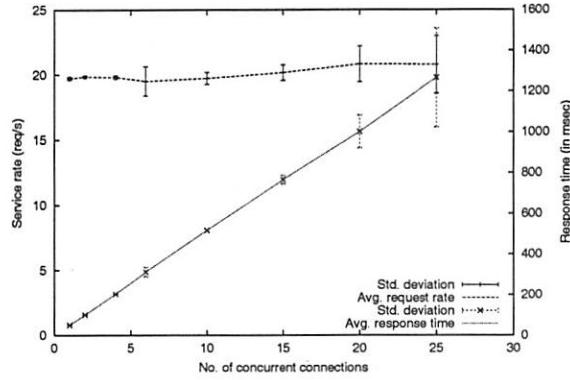


Figure 3: Peak performance: Blank coupon request.

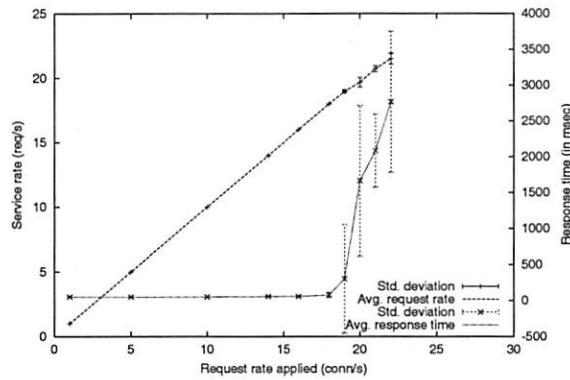


Figure 4: Typical performance: Blank coupon request.

The server system runs an Apache web server (version 1.3.12) with dynamically linked Apache_Jserv module (version 1.1.2) to process HTTP requests that invoke various Java servlets. We used Sun JDK 1.2.2 with user-level threads and no just-in-time (JIT) compilation. It runs a MySQL database server (version 3.23.27) for serving database requests. It also runs the coupon mint component, the manufacturer component, the coupon storage provider component and the retailer component. We used the cryptlib encryption toolkit² for performing cryptographic operations in C.

The client system runs httpref [18], a widely used Web performance measurement tool. Since httpref does not support SSL, it was disabled during the measurements. To test the system performance, Apache web server was configured to support persistent connections with infinite number of requests on each persistent connection, with number of start

²<http://www.cs.auckland.ac.nz/~pgut001/cryptlib/>

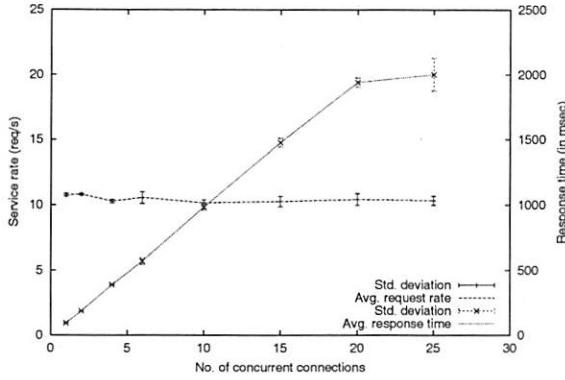


Figure 5: Peak performance: Verification request.

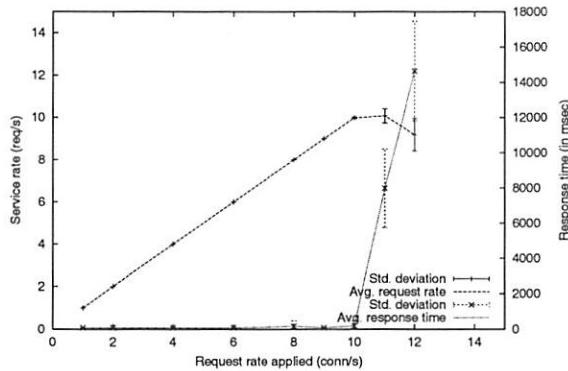


Figure 6: Typical performance: Verification request.

servers as 20 and the maximum number of simultaneous connections as 20. This is to limit the number of Apache processes as there is one process per active connection. The Apache-Jserv module was configured to disable authentication of the requesting client, auto-reloading of classes upon modification and logging except for error messages. These are factors affecting Apache-Jserv performance. To optimize database access time, a set of database connections were opened at servlet initialization. Successive requests obtain a free connection from the pool of database connections and return it to the pool, after the request is processed.

The tests were conducted for main functions of the coupon mint, namely the blank coupon generation and verification request.

For each of the messages, httperf sends appropriate number of HTTP requests so that it runs for approximately 150 seconds. The following tests were

conducted for each of the messages.

Peak performance: To determine the peak performance of the server. In this test, the client repeatedly sends HTTP requests, one request at a time, on one or more persistent HTTP/1.1 connections(s). The number of simultaneous persistent connections is increased, starting from 1, till the server saturates. The requests are sent repeatedly on all the simultaneous connections. This test should indicate the peak performance of the server since the connection setup overheads are eliminated.

Typical performance: To determine the maximum request rate which the server can handle before it reaches saturation when each request is sent on a separate connection. In this test, the client repeatedly sends HTTP requests, opening a new connection for each request. The request rate is increased, starting from 1, till the server saturates. This test does not pipeline requests on a persistent HTTP connection.

6.2 Results

Twenty five runs of the above described tests, for each of the messages, were conducted. The results were measured as the response time per request and the observed service rate i.e., the total number of requests sent divided by the total test time. Figures 3 to 6 depict the mean and the standard deviation of the results obtained.

Figure 3 shows the plot of the response time and the observed request rate with the number of simultaneous persistent HTTP connections for the blank coupon request. As the number of simultaneous connections is increased, the observed service rate remains constant with an approximate value of 20 and the response time increases linearly. This shows that the server can easily support 25 parallel simultaneous connections without getting overloaded.

Figure 4 shows the plot of the response time and the observed service rate with the request rate, for the blank coupon request. For loads of 19 requests/second or less, the server is not overloaded. The observed request rate is equal to the rate at which the requests are sent and the response time is constant. As the load increases above 19 requests/second the server begins to saturate, the response time increases exponentially and the ob-

Operation	Max. number of operations per second
Digital Signatures	86.4
Keyed Hash	3442.0
Web service (using servlets)	120.0
Database (issue)	203.3
Database (verification)	239.8

Table 1: Performance of individual operations.

served request rate levels off.

Figure 5 and 6 shows similar plot of the response time and the observed service rate for the verification request. Here the saturation occurs at the service rate of 10 requests per second. The performance for *verification requests* is lower because this message requires more cryptographic operations (a signature verification and another signature).

6.3 System Scalability

Every year 5-7 billion traditional paper coupons are redeemed in the USA [3]. The performance requirements for a real coupon mint, serving one billion coupons per year, with a peak-to-mean load ratio of 100, is about 3000 verification requests per second. On a PC implementation, the coupon mint was able to support up to 19 blank coupon issue requests per second and about 10 verification requests per second.

To understand the bottleneck in the system, we divided the system code into the following four logical components: code for cryptographic operations, code for database operations, code for web service related operations, and the prototype electronic coupons code. We wrote small benchmark programs to individually measure the performance of each of the standard components on the test system. The results are summarized in Table 1.

Carrying out a digital signature using cryptlib with a key size of 1024 bits takes an average of 11.58 ms per signature, whereas a keyed hash function (HMAC-SHA1) on a data of same size using a key of same size takes 290.5 μ sec per hash. In the prototype implementation, digital signatures are used to generate unforgeable blank coupons. Therefore, each blank coupon request requires one digital sig-

nature and each verification request requires two digital signatures. If keyed hash function is used (as explained in Section 3) instead of digital signatures, this can be reduced to one keyed hash operation for blank coupon issue and one keyed hash and one digital signature for verification request.

We wrote a simple servlet that outputs “Hello World” when invoked. Using a method similar to that used to measure the performance of the prototype system, we measured the performance of this servlet. With this servlet, the test system was able to serve up to 120 requests per second.

To measure the performance of the database system, we wrote sample programs that perform database operations equivalent to those performed during blank coupon issue and blank coupon verification. Table 1 indicates that the “issue program” is able to perform 203.3 operations per second and the “verification program” is able to perform 239.8 operations per second.

Digital signature is the primary bottleneck that limits the system performance to 86.4 request per second even if the other overheads are optimized to take nearly zero time. One method to speedup this architecture is to use multiple processors to serve the requests arriving in parallel. There are two main issues while parallelizing any program: efficient distribution of the load among different parallel machines, and taking care of data dependencies between different machines.

Fortunately for our architecture the data dependencies are very minimal and therefore it seems amenable to a large-scale distributed implementation. A single coupon mint may deploy several independent subsystems for generating and verifying blank coupons. Each subsystem has its own local database, uses its own key for generating the unforgeable blank coupons, but uses a common coupon mint key to sign the certificate of use. Now, each of these subsystem can function independently, provided the verification requests are always sent to the subsystem that generated the blank coupon corresponding to the request. Thus, while generating a blank coupon, each of these subsystems puts a verification URL in the blank coupon, so that the retailer sends the coupon to the subsystem which generated it. All the blank coupons of the same class (for early bird coupons) are also generated by the same subsystem. Thus, each subsystem functions independently without sharing its database with other

subsystems. The load may be partitioned statically across these subsystems based on the identity of the manufacturer and its specific promotion.

Therefore, it seems possible to incrementally scale the coupon mint system by adding independent subsystems as the requirement grows. According to a highly conservative estimate, a production system handling a significant fraction of coupons redeemed in the USA (if all of them were to be converted into electronic coupons) will require a farm of 200 to 400 PC-like systems to deliver adequate performance. This number is expected to go down significantly after the removal of prototyping inefficiencies and with constant advances in processor technologies.

7 Conclusions

In this paper we presented an architecture for secure generation and verification of a variety of electronic coupons. The architecture supports manufacturer and store coupons, targeted and untargeted coupons, limited and unlimited distribution coupons, and variable value coupons of different kinds (early bird, aging, growing, random value coupons).

Separation of coupon issue and distribution from online coupon verification is an important part of this architecture. Thus, individual manufacturers can issue and distribute electronic coupons without requiring an online coupon verification infrastructure of their own. The coupon verification is done using the services of an independent third party coupon mint provider. In this way, different manufacturers can implement their own promotion policies through a variety of available couponing mechanisms.

We have prototyped the electronic coupon system in Java. We briefly described the details of our prototype implementation. We also discussed the design of the retailer component which needs to be integrated with the order processing modules of the online stores.

We reported some preliminary performance measurements on the prototype system. While the performance of the prototype system is limited, we suggested some ways to scale the system for significantly higher performance. We therefore believe that, with appropriate hardware infrastructure it is

possible to develop a system which can give a performance required in a production environment.

8 Acknowledgments

Special mention goes to Alok Aggarwal who inspired the authors to carry out this work. We also thank the anonymous USENIX referees for their meticulous and insightful comments on the paper.

References

- [1] P. Kotler, *Marketing Management - Analysis, Planning, Implementation and Control*, Prentice Hall, 1991.
- [2] M. Kumar, A. Rangachari, A. Jhingran, and R. Mohan, "Sales promotion on the Internet," in *3rd USENIX Workshop on Electronic Commerce*, (Boston, MA, USA), pp. 167–176, August-September 1998.
- [3] Joint Industry Coupon Committee, *Coupons: A Complete Guide*. Grocery Manufacturers of America, 1998, URL: www.gmabrands.com.
- [4] S. Brands, "Untraceable off-line cash in wallets with observers," *Advances in Cryptology CRYPTO'93*, pp. 302–318, 1993.
- [5] D. Chaum, A. Fiat, and M. Naor, "Untraceable electronic cash," *Advances in Cryptology CRYPTO '88*, pp. 319–327, 1988.
- [6] N. Ferguson, "Single term off-line coins," *Advances in Cryptology - EUROCRYPT '93*, pp. 318–328, 1993.
- [7] M. K. Reiter, V. Anupam, and A. Mayer, "Detecting hit shaving in click-through payment schemes," in *3rd USENIX Workshop on Electronic Commerce*, pp. 155–166, August 1998.
- [8] V. Anupam, A. Mayer, K. Nissim, B. Pinkas, and M. K. Reiter, "On the security of pay-per-click and other web advertising schemes," in *8th International World Wide Web Conference*, May 1999.
- [9] NIST, "The digital signature standard," *Communications of the ACM*, vol. 35, July 1997.

- [10] R. L. Rivest, A. Shamir, and L. A. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.
- [11] O. Kommerling and M. G. Kuhn, "Design principles for tamper-resistant smartcard processors," in *USENIX Workshop on Smart-card Technology*, (Chicago, Illinois, USA), May 1999.
- [12] S. Bakhtiari, R. Safavi-Naini, and J. Pieprzyk, "Keyed hash functions," in *Cryptography: Policy and Algorithms Conference*, pp. 201–214, Springer-Verlag, LNCS 1029, July 1995.
- [13] C. Adams and S. Farrell, "Internet X.509 public key infrastructure certificate management protocols," RFC 2510, March 1999.
- [14] T. Bray, J. Paoli, and C. M. Sperberg-McQueen, "Extensible markup language (XML)," World Wide Web Consortium Recommendation REC-xml-19980210.
- [15] T. Dierks and C. Allen, "The TLS protocol version 1.0," RFC 2246, January 1999.
- [16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," RFC 2616, June 1999.
- [17] D. Connolly and L. Masinter, "The 'text/html' media type," RFC 2854, June 2000.
- [18] D. Mosberger and T. Jin, "httperf: A tool for measuring web server performance," in *WISP*, (Madison, WI), pp. 59–67, June 1998.

Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML

Don Davis*

Trust, but verify. – Russian proverb

Abstract

Simple Sign & Encrypt, by itself, is not very secure. Cryptographers know this well, but application programmers and standards authors still tend to put too much trust in simple Sign-and-Encrypt. In fact, every secure e-mail protocol, old and new, has codified naïve Sign & Encrypt as acceptable security practice. S/MIME, PKCS#7, PGP, OpenPGP, PEM, and MOSS all suffer from this flaw. Similarly, the secure document protocols PKCS#7, XML-Signature, and XML-Encryption suffer from the same flaw. Naïve Sign & Encrypt appears only in file-security and mail-security applications, but this narrow scope is becoming more important to the rapidly-growing class of commercial users. With file- and mail-encryption seeing widespread use, and with flawed encryption in play, we can expect widespread exposures.

In this paper, we analyze the naïve Sign & Encrypt flaw, we review the defective sign/encrypt standards, and we describe a comprehensive set of simple repairs. The various repairs all have a common feature: when signing and encryption are combined, the inner crypto layer must somehow depend on the outer layer, so as to reveal any tampering with the outer layer.

1 Introduction

Since the invention of public-key cryptography, cryptographers have known that naïve combinations of encryption and signature operations tend to yield insecure results [1, 2]. To guarantee good security properties, carefully designed security protocols are necessary. However, most security protocols of the past 25 years have focused on securing network connections, and relatively simple file-encryption problems have received surprisingly little attention from protocol designers.

*Affiliations: Shym Technology, 75 Second Ave. Suite 610 Needham, MA 02494; Curl Corp., 400 Technology Sq., Cambridge, MA 02139; ddavis@curl.com, don@mit.edu

Users and programmers prefer to think about security by analogy with familiar symmetric-key “secret codes.” For mail-handling and file-handling, security designers have relied heavily on simple asymmetric encryption and signing, rather naïvely combined. Naïve sign & encrypt has surprisingly different security semantics from symmetric encryption, but the difference is subtle, perhaps too subtle for non-specialist users and programmers to grasp. Indeed, for senders, sign-and-encrypt guarantees the same security properties as symmetric-key cryptography gives. With both types of crypto, the sender is sure that:

- The recipient knows who wrote the message; and
- Only the recipient can decrypt the message.

The difference appears only in the *recipient’s* security guarantees: the recipient of a symmetric-key ciphertext knows who sent it to him, but a “simple sign & encrypt” recipient knows only who *wrote* the message, and has no assurance about who *encrypted* it. This is because naïve sign & encrypt is vulnerable to “surreptitious forwarding,” but symmetric-key encryption is not. Since users always will assume that sign & encrypt is similar to symmetric-key “secret codes,” they will tend to trust naïve sign & encrypt too much.

The standards that exist for simple file-encryption, chiefly PKCS#7 [23] and S/MIME [20], tend to allow secure Sign & Encrypt implementations (i.e., such as would prevent surreptitious forwarding), but surprisingly, these file-security standards don’t require fully-secure implementation and operation. Similarly, some important new security standards, such as the XML¹ security specifications [6, 26], offer only low-level “toolbox” APIs. Too often, both the established standards and the new ones allow insecure yet compliant implementations. Application programmers need more security guidance than these “toolbox” APIs offer, in order to build effective security into their applications. Without such guidance, programmers tend to suppose incorrectly that simply signing and

¹Extended Markup Language

then encrypting a message or a file will give good security.

The limitations of naïve sign & encrypt probably were well-known to the designers of all of the standards we discuss here (see § 4.6). The standards authors assumed, sometimes explicitly and sometimes implicitly, that applications programmers and end-users would understand that naïve sign & encrypt is not a complete security solution. Application programmers were expected to know how to bolster each standard's sign & encrypt feature with other protocol elements. At the same time, end-users were expected to make careful security judgments about any application they might use, so as to use the application's security features correctly, and so as not to over-rely on a product that offers only limited security. The standards authors' expectations may have been realistic ten years ago, before Everyman and the Acme Boot-Button Co. began using the Internet. It seems unfair to fault the standards designers for insufficient prescience, but now, these expectations are hopelessly outdated, and those standards cannot serve end-users well.

1.1 Surreptitious Forwarding

Why is naïve Sign & Encrypt insecure? Most simply, S&E is vulnerable to “surreptitious forwarding.” Alice signs & encrypts for Bob’s eyes, but Bob re-encrypts Alice’s signed message for Charlie to see. In the end, Charlie believes Alice wrote to him directly, and can’t detect Bob’s subterfuge. Bob might do this just to embarrass Alice, or Charlie, or both:²

$$A \rightarrow B : \{\{“I \text{ love you”}\}^a\}^B \quad (1)$$

$$B \rightarrow C : \{\{“I \text{ love you”}\}^a\}^C \quad (2)$$

Here, Bob has misled Charlie to believe that “Alice loves Charlie.” More serious is when Bob undetectably exposes his coworker Alice’s confidential information to a competitor:

$$A \rightarrow B : \{\{“sales plan”\}^a\}^B \quad (3)$$

$$B \rightarrow C : \{\{“sales plan”\}^a\}^C \quad (4)$$

In this case, Alice will be blamed conclusively for Bob’s exposure of their company’s secrets.

Further, when Alice signs a message to Bob, Alice may be willing to let Charlie see that message, but

²Notation: “A” is Alice’s public key, and “a” is her private key. Thus, $\{msg\}^A$ is an encrypted ciphertext, and $\{msg\}^a$ is a signed message. We assume that the asymmetric-key cryptosystem behaves similarly to RSA [21], so that a signature is a private-key encryption.

not to *sign* the same message for Charlie:

$$A \rightarrow B : \{\{“I.O.U. \$10K”\}^a\}^B \quad (5)$$

$$B \rightarrow C : \{\{“I.O.U. \$10K”\}^a\}^C \quad (6)$$

If every user could be relied upon to understand that Sign & Encrypt is vulnerable to surreptitious forwards, then Alice wouldn’t have to worry about Bob forwarding her message to Charlie. But in reality, when Charlie gets Alice’s message via Bob, Charlie very likely will assume that Alice sent it to him directly. Thus, even if Alice doesn’t care whether Bob divulges the message, she may be harmed if Bob is able to forward her signature surreptitiously.

1.2 Don’t Sign Ciphertexts

Interestingly, naïve Encrypt-then-Sign isn’t any better than Sign & Encrypt. In this case, it’s easy for any eavesdropper to replace the sender’s signature with his own, so as to claim authorship for the encrypted plaintext:

$$A \rightarrow| B : \{\{“my idea”\}^B\}^a \quad (7)$$

$$C \rightarrow B : \{\{“my idea”\}^B\}^c \quad (8)$$

Note that Charlie has to block Bob’s receipt of Alice’s original message, before sending the re-signed ciphertext.

Another problem with Encrypt-then-Sign arises, when Alice uses RSA or El Gamal encryption. In a sequel to Abadi’s “Robustness Principles” paper [1], Anderson showed that Encrypt&Sign is dramatically weaker than had been thought [2]. Suppose Alice uses RSA keys to send Bob an E&S message:

$$A \rightarrow B : \{\{msg\}^B\}^a \quad (9)$$

Then Bob can pretend that Alice encrypted and signed an *arbitrary* message msg' , of his choice. To alter Alice’s plaintext, Bob uses the factors of his own RSA modulus n_B to calculate the discrete logarithm x of Alice’s message msg , using as base Bob’s arbitrary message msg' :

$$\{msg'\}^x = msg \pmod{n_B} \quad (10)$$

Now, Bob needs only to certify (xB, n_B) as his public key, in order to make Alice’s original ciphertext signature valid for Bob’s new encryption $\{msg'\}^{xB}$:

$$B \rightarrow B : \{\{msg'\}^{xB}\}^a \quad (11)$$

Anderson's attack has two minor limitations:

- Each modulus factor must be short enough (~120 digits, or ~400 bits) to allow a discrete-log calculation [13];
- Bob's new public exponent xB will be obviously unusual, in that it will be a full-length bitstring, instead of the usual small integer value.

So, it might seem that Alice should be safe from this attack, as long as Bob's public key B is substantially longer than 240 digits (800 bits). Unfortunately, Alice cannot tell, without factoring Bob's RSA key-modulus, whether Bob used three or more prime factors to prepare his RSA key-pair [22]. If Bob has a large-modulus key-pair made up from several small factors, then Alice's naïve use of Encrypt & Sign would still leave her vulnerable to Bob's substituted-ciphertext attack.

Thus, whenever we want to sign a ciphertext, Anderson's attack forces Alice to sign, along with her ciphertext, either the plaintext itself or Bob's public key B :

$$A \rightarrow B : \{\{msg\}^B, \#msg\}^a \quad (12)$$

$$A \rightarrow B : \{\{msg\}^B, \#B\}^a \quad (13)$$

The two formats offer different advantages: signing the plaintext alongside the ciphertext gives non-repudiation, while signing the encryption key is more easily understood as a defense against Anderson's attack. In either format, Bob can still alter B and msg simultaneously, so that $\{msg'\}^{B'}$ is the same as Alice's ciphertext $\{msg\}^B$. But, in order to preserve Alice's signature, Bob now also has to choose msg' to have the same hash value as the one Alice signed, and this is too difficult.

Of course, Encrypt-then-Sign isn't very useful anyway, because only the illegible ciphertext, not the plaintext, would be non-repudiable. In what follows, for simplicity, we'll mostly ignore Encrypt & Sign, and we'll concentrate on analyzing and fixing Sign & Encrypt's defects.

1.3 Purpose of the Paper

This paper intends to fill the gap between the "do-it-yourself" toolbox APIs and the "out-of-the-box" secure-networking standards:

- Section 2 describes the problem's technical and social scope,
- Section 3 analyzes the problem cryptographically,

- Section 4 reviews several standards that accept naïve Sign / Encrypt as secure, and
- Section 5 presents a comprehensive variety of simple solutions.

Our goal is to help security standards offer a variety of *secure* ways to sign and encrypt messages. Application programmers should not be constrained by "one size fits all" protocols, but they also shouldn't have to understand the nuances of cryptographic design.

2 Problem Scope

Why is this old and easy problem worth discussing at this late date? Though designing a secure Sign & Encrypt protocol is easy for cryptographers, it's a different class of engineer who faces this problem nowadays. Application programmers have to rely on crypto vendors and crypto standards, in order to learn how to write crypto applications. Unfortunately, the vendors and standards have left unintended a big gap in their support for application programmers. Current security standards don't give application programmers a simple recipe for file-encryption problems.

2.1 Technical Scope

Secure session protocols have attracted a lot of research attention, and several effective session-security protocols have been standardized, so naïve Sign & Encrypt is not a problem in session security. Session-security standards, like Kerberos [19], TLS [5], and SET [28], give straightforward, out-of-the-box solutions. For files and one-way messaging, though, current security standards give developers only a kind of "toolbox" support, with a variety of security options, but with no clear or firm guidance about how to combine the options to make Sign & Encrypt an effective security solution. Providing only toolbox-style cryptographic protocols is appropriate for a low-level mechanism like IPSEC [12], but for user-visible applications like secure e-mail, programmers need "turnkey" cryptography, not only cryptographic toolkits.

Thus, naïve Sign & Encrypt has come to characterize file-handling and e-mail security applications. PKCS#7 [23], CMS³ [9], S/MIME [20], and PGP⁴ [29], all suffer from this defect. Further, the W3C's⁵ XML-Signature & XML-Encryption Working Groups have explicitly set themselves the task

³Cryptographic Message Syntax.

⁴Pretty Good Privacy

⁵The World Wide Web Consortium, see <http://w3.org>.

of supplying XML with S/MIME-style security. The demand for simple file-security and message-security is big and growing, so widespread use of these naïve Sign & Encrypt security models will lead to widespread exposures.

2.2 Social Scope

Increasingly, secure applications are being designed and built by application programmers, not by cryptographers. Several factors have obliged mainstream application programmers to undertake public-key protocol design:

- Commercial PKI is in widespread deployment;
- Secure networking standards don't address file-encryption;
- Demand for cryptographers greatly exceeds the supply.

So, when application programmers need file-encryption help, they can seek help from crypto vendors and from crypto standards. Unfortunately, the vendors and the standards both offer either high-level secure connections, or low-level “toolkit” mechanisms. Neither offering makes file-encryption easy. The available standards specifications for file-encryption intend to support security applications, but the specifications tend to standardize only low-level APIs for cryptographic primitives, so as to leave designers as much flexibility as possible.

3 Defective Standards

The delicacy of naïve Sign & Encrypt is a well-known issue in S/MIME. Similar flaws appeared in 1986 in the first version of the PGP message-format [30], and in 1988 in X.509v1 [14]. X.509's flaw was discovered in 1989 by Burrows et al. [4], and a correct repair was proposed in 1990 by I'Anson and Mitchell [11]. Unfortunately, more recent workers have failed to apply I'Anson's simple repair correctly; PEM and PKCS#7 suffer from a defective version of I'Anson's repaired Sign & Encrypt, and the same defect is now codified by S/MIME. In parallel with these developments, PGP independently retained the same naïve Sign & Encrypt defect. The current protocols' flaw is substantially similar to the original flaws in X.509 and PGP. So, the historical flow of inheritance is:

- Zimmermann described a naïve RSA-based Sign & Encrypt protocol, which later became PGP;
- X.509v1 codified a flawed, naïve Encrypt & Sign, independently of PGP;

- Burrows et al. and I'Anson described a workable Sign & Encrypt protocol for X.509;
- PEM applied X.509's cryptography to e-mail transport, using naïve S&E instead of I'Anson's repaired S&E;
- Three standards extended and generalized PEM:
 1. MOSS extended PEM to support MIME-encoded e-mail, by adding naïve Sign & Encrypt for e-mail attachments;
 2. PKCS#7 generalized PEM to non-mail file-handling applications, but preserved the S&E flaw intact;
 3. CMS and S/MIME carried PKCS#7's generality and the flawed S&E back to the e-mail community.
- Today, the nascent XML security standards expressly intend to support naïve Sign & Encrypt.

These relationships aren't as complicated as they look, because MOSS, PKCS#7, and S/MIME are all descended from PEM, and through PEM from X.509, while PGP and XML are completely independent efforts.

In the rest of this section, we discuss the defective standards in the chronological order listed above.

3.1 PGP and OpenPGP

PGP is similar to PEM and simpler than S/MIME, in that PGP provides only three security options: Sign, Encrypt, and Sign & Encrypt. Of these security options, we are only interested in PGP's Sign & Encrypt (we will discuss only Sign & Encrypt in the other standards' subsections, too).

PGP's message-format had several similarities with later features of PEM and S/MIME:

- symmetric-key encryption for message bodies;
- unformatted message-bodies;
- independent crypto layers.

In our discussion, we'll omit PGP's use of symmetric-key ciphers for bulk encryption, because it is irrelevant to our surreptitious forwarding attack.

PGP's strongest security option is naïve Sign & Encrypt, so PGP is vulnerable to surreptitious forwarding:

$$A \rightarrow B : \{ \{ "The deal is off." \}^a \}^B \quad (14)$$

$$B \rightarrow C : \{ \{ "The deal is off." \}^a \}^C \quad (15)$$

Here, Alice has cancelled a deal with Bob, so Bob gets even with her later, by re-encrypting and redirecting Alice’s signed message to her next business partner, Charlie.

Note that PGP’s plaintext message-bodies are unformatted, containing no names for the sender or recipient. Because PGP doesn’t allow formatted message bodies, an extra signature layer, or signed attributes, PGP doesn’t admit any of the protocol repairs we describe below for S/MIME and PKCS#7 (see §§ 3.5, 3.6, & 5.1).

3.2 X.509, Version 1

The first version of X.509 included a simple protocol for secure message-exchange, employing secure message “tokens” with the following structure:

$$A \rightarrow B : \{Bob, \#msg, \{msg\}^B\}^a \quad (16)$$

Burrows et al. [4] pointed out that C could readily replace A’s signature with his own, leading B to attribute A’s message to C:

$$C \rightarrow B : \{Bob, \#msg, \{msg\}^B\}^c \quad (17)$$

(See also Eqn.7). So, I’Anson and Mitchell [11] offered a repaired token-structure for X.509:

$$A \rightarrow B : \{\{(Bob, msg)\}^a, msg\}^B \quad (18)$$

Unfortunately, I’Anson’s cryptographic notation was hard to understand,⁶ and his text didn’t emphasize exactly what made his corrected token secure:

This modification involves no additional effort as far as token construction is concerned, and it is simply to require that the encryption of enc-Data is done *after* the signature operation instead of before.

I’Anson’s text incorrectly implied that he had only replaced E&S with S&E. In fact, his repair worked only because he made Alice sign her recipient’s name, *Bob*, along with her message. This signed name proved Alice’s intent to write for Bob. If Alice’s signature hadn’t included Bob’s name, then I’Anson’s new token would have been just a naïve Sign & Encrypt, fully vulnerable to surreptitious forwarding.

Clearly, I’Anson’s paper influenced the early PKI standards community, because PKCS#1 and various later RFCs cited the paper. Though PEM and later mail standards didn’t cite I’Anson, they followed his paper’s advice: PEM, PKCS#7, and CMS provided

⁶In Eqn.16, we’ve simplified the X.509 token’s structure, by leaving out various nonces and other parameters.

Sign & Encrypt as a basic operation, and S/MIME explicitly deprecated Encrypt & Sign. We suggest that had I’Anson explained the necessity of signing the recipient’s name, the later standards would have used Sign & Encrypt correctly.

Note that X.509’s original Encrypt & Sign token (cf. Eqn. 16, above) could have been fixed without signing first, by the simple addition of the sender’s name, similar to I’Anson’s signed recipient-name:

$$A \rightarrow B : \{\#msg, \{Alice, msg\}^B\}^a \quad (19)$$

This repair, like I’Anson’s, blocks Burrow’s signature-replacement attack (cf. Eqn. 17), because Bob can now detect Charlie’s replacement: if the signer’s certificate doesn’t match Alice’s name inside the plaintext, then Bob can conclude that the message was tampered with. This repair also repairs Encrypt & Sign’s non-repudiation problem, since Alice signs her plaintext explicitly. Finally, this repair also blocks Anderson’s plaintext-replacing attack (see § 1.2).

3.3 PEM

Privacy-Enhanced Mail was the first notable secure-email standard for the Internet. PEM was designed and specified in the late 1980’s and early 1990’s [15]. The first version of PEM relied exclusively on symmetric-key cryptography, but as X.509’s PKI specification settled, later versions of PEM increasingly emphasized public-key cryptography. It seems likely that PEM’s over-reliance on naïve Sign & Encrypt led PEM’s descendants MOSS, PKCS#7, and S/MIME to follow suit. Indeed, the later specifications tried hard to support backward-compatible interoperation with PEM.

For our purposes, PEM provides essentially only two variants of mail security; a message can be signed only, or it can be signed and then encrypted. Like PGP, and like PEM’s descendants PKCS#7, CMS, and S/MIME, PEM applies its signature and encryption steps to the message-body, i.e., *not* to the SMTP header, the “From: / To:” header, or to the “encapsulated header,” which carries a PEM message’s keys and names. PEM has no notion of signing or authenticating ancillary attributes, and also doesn’t support extra crypto layers, so the repairs we discuss below for S/MIME and PKCS#7 (see §§ 3.5 & 3.6) won’t work for PEM. To prevent surreptitious forwarding, a PEM message’s author would have to include the recipient’s name directly in the message-body. Of course, it could be very difficult for the receiving PEM mail-client to find the recipient’s name in the body, so as to check automatically for surreptitious forwarding.

Today, PEM is not widely used, and PEM’s vulnerability to surreptitious forwarding is mostly just

a matter of historical interest. But PEM's accomplishment and influence were great, because PEM successfully achieved platform-independent cryptographic interoperation, at a time when the still-new Internet was a much more heterogenous affair than it is today.

3.4 MOSS

MOSS extended PEM's cryptography in three principal ways:

1. By adding cryptographic support for MIME-formatted multipart messages (popularly known as attachments);
2. By allowing encryptions and signatures to be applied in any order, like S/MIME;
3. By decoupling secure mail from the monolithic X.500 public-key infrastructure, which had failed by the mid-1990's.

Like PEM, MOSS was eclipsed by S/MIME and by PGP, and is little heard-of today.

MOSS had another feature, one very valuable for our purposes: unlike the other secure e-mail protocols, MOSS explicitly provided by default for a sender Alice to be able to sign her message-header, along with her message-body. MOSS is the only e-mail standard that gives users such an out-of-the-box mechanism for signing the recipient-list. (S/MIME's ESS feature did allow header-signing, but this was explicitly intended as a link-oriented security feature for military mail servers. See the discussion of ESS, in the last half of §3.6.)

Header-signing was easy for MOSS to provide, because MOSS treated the header as just another "part" in the message. If Alice's MOSS message carried her signature and encryption on *both* the message-body and the message-header, Alice's MOSS message and her recipients would be fairly well-protected against surreptitious forwarding. Unfortunately, MOSS made header-signing an optional feature, and the MOSS RFCs don't discuss why header-signing is valuable. As specified, MOSS is as vulnerable to our attack as the other e-mail protocols are.

It's worth noting that even when Alice does choose to sign MOSS's header, MOSS's cryptography still relies too much on Bob's sophistication about e-mail security:

- When Bob receives Alice's MOSS message, he does have to read Alice's signed header, so as to make sure that Alice intended to send the message to him.

- Further, when Alice's cc-list is long, Bob still has to read the signed header, but this step is neither as automatic nor as reliable as one would like.
- Finally, if Alice's mail-client doesn't bother to sign her mail-headers, Bob probably won't notice, so he'll still be vulnerable to surreptitiously-forwarded messages.

All of these issues would vanish, if MOSS had made header-signing mandatory. Bob's e-mail reader presumably would automatically scan the header, looking for Bob's decryption-key's "name form," and if this search were to fail, the MOSS mail-reader would raise an error-message warning Bob.

3.5 PKCS#7

PKCS#7 was created as a file-oriented adaptation and extension of PEM's platform-independent cryptographic features. Accordingly, PKCS#7 inherited naïve Sign & Encrypt from PEM.

In order to bolster PKCS#7's Sign & Encrypt security, how might a PKCS#7 author securely attach names to a file or message? Each PKCS#7 message has SignerInfo and RecipientInfo fields, but the specification does not allow these fields to be signed or encrypted. PKCS#7 does provide for application-defined "authenticated attributes," though, so a PKCS#7 application could create a signed "To-List" attribute, so as to prove to recipients that they are the author's *intended* recipients. But crucially, PKCS#7 does not require or even suggest that for effective security, such a signed "To-list" should accompany the message. Further, PKCS#9 [24], which defines various attributes for PKCS#7 messages, similarly fails to provide any attributes for holding senders' or recipients' names.

Note also that in order to use authenticated attributes for repairing PKCS#7 Sign and Envelope, one must separately apply the signature and encryption steps, instead of using the Signed-and-Enveloped construct. This is because the combined construct doesn't support attributes at all [23]:

Note. The signed-and-enveloped-data content type provides cryptographic enhancements similar to those resulting from the sequential combination of signed-data and enveloped-data content types. However, since the signed-and-enveloped-data content type does not have authenticated or unauthenticated attributes, nor does it provide enveloping of signer information other than the signature, the sequential combination of signed-data and enveloped-data content

types is generally preferable to the Signed-AndEnvelopedData content type, except when compatibility with the ENCRYPTED process type in Privacy-Enhanced Mail is intended.

Thus, for PKCS#7's simple Signed-and-Enveloped message, the protocol affords no cryptographically secure naming. The only way a Signed-and-Enveloped recipient can know that he is intended to see the message, and that no surreptitious forwarding has occurred, is for the sender to include the recipient's name within the message-body.

3.6 S/MIME and CMS

S/MIME is a set of secure email standards, which specify not only how to encrypt and sign messages, but also how to handle keys, certificates, and crypto algorithms. CMS is the specification that describes the data-formats and procedures needed for encryption and signatures. CMS is mostly identical to PKCS#7, from which it descends.

The S/MIME specification itself acknowledges that CMS' Sign & Encrypt isn't very secure, but the S/MIME specification fails to discuss the main defect. Further, the document tells implementors nothing about how to shore up Sign & Encrypt. Instead, the S/MIME specification merely cautions users and implementors not to over-rely on a message's security:

1. "An S/MIME implementation MUST be able to receive and process arbitrarily nested S/MIME within reasonable resource limits of the recipient computer.
2. "It is possible to either sign a message first, or to envelope⁷ the message first. It is up to the implementor and the user to choose. When signing first, the signatories are then securely obscured by the enveloping. When enveloping first, the signatories are exposed, but it is possible to verify signatures without removing the enveloping. This may be useful in an environment where automatic signature verification is desired, as no private key material is required to verify a signature.
3. "There are security ramifications to choosing whether to sign first or to encrypt first. A recipient of a message that is encrypted and then signed can validate that

the encrypted block was unaltered, but cannot determine any relationship between the signer and the unencrypted contents of the message. A recipient of a message that is signed-then-encrypted can assume that the signed message itself has not been altered, but that a careful attacker may have changed the unauthenticated portion of the encrypted message" [sic].

– [20] Sec. 3.5, "Signing and Encrypting."

This excerpt is the S/MIME specification's only discussion of Sign & Encrypt's limitations. Several features in the excerpt deserve comment:

- Paragraph 2 presents the security issues as a tradeoff between confidentiality and ease of verification;
- Paragraph 3 hints that an attacker can replace the external signature in an encrypted-then-signed message,
- But there's no mention that sign-then-encrypt is vulnerable to surreptitious forwarding, by replacement of the outermost encryption layer. (In paragraph 3, "unauthenticated portion" seems to refer not to the unauthenticated ciphertext, but to unauthenticated plaintext.)
- The excerpt presents only the choice between signing first and encrypting first. There's no mention of repairing either option's defects.

S/MIME is flexible enough to allow the Sign & Encrypt defect to be repaired. In the specification excerpt above, the first paragraph provides that every S/MIME application must be able to process Sign/Encrypt/Signed messages and Encrypt/Sign/Encrypted messages. Either S/E/S or E/S/E suffices to reveal any alteration of the sender's crypto layers, as long as the receiving client knows how to detect the alterations (See §§ 5.2 & 5.3, below).

Note that our S/E/S double-signing only superficially resembles S/MIME's optional "triple-wrapping" feature; the two are different in mechanism and in purpose. S/MIME's Enhanced Security Services specification [7] provides specialized security-related message-attributes, in support of certain features such as signed receipts and secure mailing-lists. In order to support the ESS features, some mail servers will apply an extra signature to the ciphertext of an end-user's Signed-and-Encrypted message:

⁷The S/MIME, CMS, and PKCS#7 specification documents use the verbs "encrypt" and "envelope" interchangeably.

1.1 Triple Wrapping Some of the features of each service use the concept of a "triple wrapped" message. A triple wrapped message is one that has been signed, then encrypted, then signed again. The signers of the inner and outer signatures may be different entities or the same entity. Note that the S/MIME specification does not limit the number of nested encapsulations, so there may be more than three wrappings.

1.1.1 Purpose of Triple Wrapping Not all messages need to be triple wrapped. Triple wrapping is used when a message must be signed, then encrypted, and then have signed attributes bound to the encrypted body. Outer attributes may be added or removed by the message originator or intermediate agents, and may be signed by intermediate agents or the final recipient. [...]

The outside signature provides authentication and integrity for information that is processed hop-by-hop, where each hop is an intermediate entity such as a mail list agent. The outer signature binds attributes (such as a security label) to the encrypted body. These attributes can be used for access control and routing decisions.

Triple-wrapping allows mail servers to securely annotate messages on-the-fly ("hop-by-hop"), primarily for the benefit of other mail-servers. In contrast, in our S/E/S repair, Alice applies her outer signature, without any extra attributes, to her own Signed & Encrypted message, as the basic CMS specification allows. Similarly, only Alice's intended S/E/S recipient Bob would validate her inner and outer signatures. In sum, our S/E/S is an end-to-end security feature, while ESS uses triple-wrapping to support link-oriented security features.

Further, ESS triple-wrapping and S/E/S serve different purposes. Though the first two ESS paragraphs do mention that an end-user like our Alice might apply an outer signature herself, the ESS document gives no reason that she might do so, except to attach signed attributes to the ciphertext. The ESS document nowhere suggests that triple-wrapping might be necessary to repair a security defect in Sign & Encrypt. In fact, the ESS specification committee did *not* intend triple-wrapping to be a repair for the surreptitious-forwarding defect. Instead, the ESS specification was written to fulfill the U.S. Dept. of Defense's purchasing criteria for secure e-mail, which demanded server-oriented security features [8].

Besides S/E/S, another S/MIME repair option comes from the CMS specification, which is a core

piece of the S/MIME standards suite. Like PKCS#7, CMS provides for "signed attributes," which offer a different way to prevent crypto alterations. Suppose the sender includes a signed "To-List" attribute, and suppose the recipient knows how to process and interpret such an attribute. Then the recipient can identify who intended him to receive the message, and no attacker can profit by replacing the outer crypto layers. Unfortunately, like the PKCS#7 specification, the CMS specification does not stipulate or even suggest such naming attributes, though the specification does suggest other signed attributes.

These S/MIME repairs are cumbersome, and they only barely meet the e-mail industry's needs. Crucially, because the specification neither requires any repair, nor even mentions that some features can serve as repairs, the repairs' interpretations aren't standardized, and different vendors' S/MIME applications can't readily interoprate with full Sign & Encrypt security.

3.7 XML Security

At this writing (Spring 2001), the XML-Signatures draft specification [6] is nearing completion, and the allied XML-Encryption Working Group [26] is just starting its work. Both groups have explicitly committed to producing low-level "toolkit" specifications, which will describe how to combine basic public-key operations with a rich array of XML document-structuring features. In particular, both groups are very unwilling to stipulate any high-level security behavior, such as how to sign and encrypt with full security.

To some extent, this is proper: these standards are intended to support as broad a class of applications as possible, including document preparation and handling, financial applications, wire protocols, and potentially even intricate cryptographic security protocols. The Secure XML Working Groups say that they don't want to require secure high-level behavior in their specifications, because they don't want to constrain how low-level applications will use XML's security features. The WGs explicitly hope that a higher-level XML security specification, with out-of-the-box "idiot-proof" security, will be built someday to follow on the current WGs' specifications. But for now, certainly, the XML-Signatures draft specification is most suitable for use only by experienced security engineers and cryptographers, and not for application programmers who don't want to specialize in security.

4 Analysis

We propose that users of file-security and mail-security need simple security semantics, and that symmetric-key semantics are sufficient for most users and most applications' needs. Further, symmetric-key semantics are natural and easy for unsophisticated users to understand.

In this section, we present three overlapping views of what's wrong with naïve Sign & Encrypt. Then, we summarize and discuss several arguments in defense of the naïve Sign & Encrypt standards. Finally, we discuss how this flaw survived several standards-review committees' deliberations.

4.1 Asymmetric Security Guarantees

At first glance, naïve Sign & Encrypt seems quite secure, because message-author Alice gets the security guarantees she needs: her signature proves her authorship, and she knows who can read the message. The reader, Bob, doesn't get the same guarantees, though. He knows who wrote the message; but he doesn't know who encrypted it, and therefore doesn't know who else besides Alice has read the message. Note the asymmetry:

- When A sends B a signed & encrypted message, A knows that only B can read it, because A trusts B not to divulge the message, but –
- When B receives A's signed & encrypted message, B can't know how many hands it has passed through, even if B trusts A to be careful.

Seen this way, the flaw in naïve Sign & Encrypt is that B gets no proof that it was A who encrypted the message. In hindsight, this is obvious: public key algorithms usually don't automatically authenticate the encryptor of a message.

Certainly, in some applications, it's neither necessary nor feasible to give a recipient any assurance that only the sender has seen the message-plaintext. Thus, for example, mail-security applications do need the flexibility to waive full end-to-end symmetric-key semantics. But, whenever possible, and by default, mail- and file-security applications should give end-users easy-to-understand security guarantees.

4.2 Symmetric-Key Semantics

Users tacitly expect public-key file-encryption to offer the same security semantics that a symmetric key offers. Thus, another way to describe the Sign & Encrypt problem is that whether signing or encryption is applied first, naïve Sign & Encrypt fails to duplicate

the security meaning of a symmetric-key ciphertext. When B receives a symmetric-key ciphertext from A, B can safely assume that:

- A sent the message,
- No-one else has seen the plaintext,
- A intended B to receive the plaintext.

With naïve Sign & Encrypt, these assumptions can break down, because the recipient may have to rely on the crypto layer to supply the intended recipient's names. That is, the problem arises when:

- The message plaintexts don't mention the sender's and target's names;
- The sender's and recipient's names are important for understanding the message or its security import;
- The recipient *assumes* that the signer encrypted the message.

Under these conditions, an attacker can successfully and surreptitiously forward a naïvely signed and encrypted message.

4.3 Sign & Encrypt Must Cross-Refer

We suggest that the messaging standards all erred by treating public-key encryption and digital signatures as if they were fully independent operations. This independence assumption is convenient for writing standards and for writing software, but it is cryptographically incorrect. When independent operations are applied one on top of another, then the outermost crypto layer can undetectably be replaced, and security is weakened.

In [1], Abadi and Needham presented a simple best-practice rule for protocol design:

When a principal signs material that has already been encrypted, it should not be inferred that the principal knows the content of the message. On the other hand, it is proper to infer that the principal that signs a message and then encrypts it for privacy knows the content of the message.

In [2], Anderson and Needham presented their plaintext-substitution attack against Encrypt-then-Sign (see § 1.2), and they strengthened Abadi's prescription:

Sign before encrypting. If a signature is affixed to encrypted data, then ... a third party certainly cannot assume that the signature is authentic, so nonrepudiation is lost.

These principles were well-understood soon after X.509’s defect was discovered (if not before), and to be fair, they were published after the early versions of PEM, PKCS#7 and S/MIME were published. But PKCS#7 and S/MIME have been revised since Abadi’s and Anderson’s papers became well-known, so the updated standards could have been repaired. Nevertheless, the e-mail standards still treat the Sign & Encrypt problem as a user-interface issue: “There are security ramifications to choosing whether to sign first or encrypt first...” [20].

Though signing and encryption are not independent of one another, the defective standards treated crypto operations as independent content-transformations, converting “content” to “content.” Conceptually, this makes it easy for users and programmers to layer crypto operations in arbitrary depth and in arbitrary order. By this device, the standards authors sought to avoid constraining application developers’ designs.

With such independent operations, though, it’s hard to fulfill the recipient’s security expectations. In order to work properly together, the signature layer and the encryption layer actually must refer to one another, so as to achieve basic symmetric-key security guarantees that users expect. The recipient needs proof that the signer and the encryptor were the same person, which necessarily entails either signing the recipient’s identifier (in Sign & Encrypt), or encrypting the signer’s identifier (in Encrypt & Sign). Once such cross-references are in place, an attacker can’t remove and replace the outermost layer, because the inner layer’s reference will reveal the alteration.

In Section 5, “Repair Options,” we present five ways to give the recipient this cross-referenced proof of the encryptor’s identity. In each of these five repairs, the sender identifies the outermost operation’s key-holder, inside the innermost content, so as to bind the sender’s and recipients’ names together. For example, one repair for Sign & Encrypt puts the decrypting recipient’s name inside the signed plaintext message:

$$A \rightarrow B : \{ \{ "To : Bob", msg \}^a \}^B \quad (20)$$

This repair is straightforward for a user or an implementor to do, but it’s hard for a standards specification to stipulate that different crypto operations must be tied together like this, without breaking the full generality of the content-transformation model.

4.4 Trust and Risk

A common defense of naïve Sign & Encrypt is that users have to be careful about whom they trust, or

equivalently, that users should carefully assess risk when putting sensitive material under cryptographic protection. In this view, the recipient of a signed and encrypted message should not invest more trust in the message than the technology and the sender’s reputation can support. This argument seems very plausible, but it turns out not to address the problems with naïve Sign & Encrypt.

B has no way to gauge the risk that the message has been divulged to people unknown to A and B. To gauge the risk, B would have to know how trustworthy are the people who have surreptitiously forwarded the message along from A towards B. Thus, in general, one can’t assess the privacy of a decrypted plaintext, and shouldn’t trust its privacy, unless one knows who encrypted it. In sum: if we accept the Trust and Risk argument, then the encryption step of Sign & Encrypt is quite pointless from the receiver’s point-of-view.

4.5 Security and Ease-of-Use

Another common defense of S/MIME’s naïve Sign & Encrypt is that “Users shouldn’t trust unsigned information” about the signer’s intended recipients. This argument misses the point of S/MIME’s weakness, by supposing that users are over-relying on the unsigned SMTP header to identify the sender’s intended recipients. The users’ mistake is more subtle, though; they’re over-relying on the encrypting-key’s certificate, as a secure record of the sender’s intended recipient.

It’s unrealistic to expect today’s users to catch such a subtle point. When X.509, PEM, and S/MIME were designed, PKI users were expected to be system administrators and other fairly sophisticated users; now, though, with the modern Internet and with electronic commerce in play, we can’t expect most users to understand any cryptographic nuances at all.

A similar defense of the defective secure mail standards is that the specifications aren’t actually broken, because “Applications can and should put names into the content, if that’s what they want.” This argument assumes that application programmers shouldn’t try to incorporate cryptographic security into programs in the first place, unless they understand security and cryptography well enough to design security protocols. Further, the argument insists that no security standard can be so complete as to prevent ignorant programmers from “shooting themselves in the foot.”

A ready answer to this argument is “SSL.” The SSL specification gives fairly complete security, out-of-the-box. Further, non-specialist programmers are able to set up secure SSL connections for their appli-

cations, without having to patch the SSL protocol on their own.

4.6 How Did This Happen?

According to the authors of the PEM [16, 17], S/MIME [8, 10], and XML-Security [25] standards, those working groups explicitly discussed surreptitious forwarding, and yet deliberately left the flaw unrepaired. The committees accepted this cryptographic neglect for several reasons:

- *Optional Coverage*: All of the specifications *allow* senders to put the recipient’s name, or the whole mail header, into the message-body before signing. In addition, some protocols explicitly provide an optional mechanism for signing the mail header or the recipient-list.
- *Contextual Repair*: In the same way, the PEM committee’s discussion explicitly decided that the message’s context would *usually* solve the problem. For example, Alice’s signed “Dear Bob” salutation would reveal any re-encryption.
- *Out of Scope*: The PEM committee noted that surreptitious forwarding is a type of replay, and that no e-mail mechanism can prevent e-mail replay. Thus, to the PEM committee, it seemed inappropriate to worry about surreptitious forwarding of signed-and-encrypted mail.

More recently, the XML-Signature and XML-Encryption working groups explicitly decided, from the outset of their work, to emulate S/MIME’s security. Both groups decided not to address S/MIME’s and PKCS#7’s vulnerability to surreptitious forwarding, for three related reasons:

1. XML-Signature and XML-Encryption are explicitly low-level protocols. Thus, the XML-security standards mustn’t force higher-level protocols to follow a particular cryptographic model.
2. The W3C intends that for XML documents, format specifications and semantics specifications should generally be kept separate. Accordingly, surreptitious forwarding, being an issue of Sign & Encrypt “semantics,” should be treated in a separate XML Security Semantics specification.
3. A document-format working group shouldn’t try to resolve questions about minute details of cryptographic implementation, because such discussions invariably become time-wasting “ratholes.”

Thus, the XML-Security working groups seem to intend their specifications to be accepted as strictly “low-level” cryptographic primitives. It’s hard, though, to reconcile this “low-level” label with these working groups’ early proposal to emulate S/MIME, since S/MIME claims to offer high-level, comprehensive, and secure messaging.

It’s hard to blame the secure-mail standards groups for having made a cryptographic mistake. Clearly, they all worked in good faith to promote secure and usable technologies. Further, it’s important to acknowledge how hard it is to write networking standards in general, and mail-related standards in particular. As hard as it is to design cryptographic security protocols, cryptographic difficulty is only a formal or mathematical affair, and is very different from the difficulty of designing workable networking protocols for real-world deployment. In any design of a concrete security protocol, many hard problems have to be solved simultaneously, including:

- Flexibility for application programmers;
- Flexibility for network admins and sys-admins;
- Interoperation with other protocols;
- OS platform differences;
- Scaling;
- Server statelessness;
- Exportability;
- Time-to-market.

Clearly, each of the secure e-mail standards committees tried to codify a cryptographically correct protocol. The worst that can be said of these working groups is that they underestimated the subtlety of adding cryptography to their already-burdened portfolio.

5 Repair Options

We present five independent and equivalently-secure ways to fix the naïve Sign & Encrypt problem:

1. Sign the recipient’s name into the plaintext, or
2. Encrypt the sender’s name into the plaintext, or
3. Incorporate both names; or
4. Sign again the signed-&-encrypted message; or
5. Encrypt again the signed ciphertext.

In each case, the signing layer and the encryption layer become interdependent, binding the sender’s name, in one layer, to the recipient’s name in the other layer. Any one of these alternatives suffices to establish that Alice authored both the plaintext and the ciphertext. Note though that an effective security standard should require not only that the author must *provide* one of these five proofs, but also that the recipient must *demand* some such proof as well. That is, if a naïve Sign & Encrypt message arrives without proof that the signer and encryptor were the same person, then the application software should warn the recipient that the message’s privacy and/or authenticity are suspect.

5.1 Naming Repairs

Perhaps part of the reason naïve Sign & Encrypt seems secure is that with many common payload messages, S&E is secure. For example, even if Alice just signs and encrypts the text “Dear Bob, The deal is off. Regretfully, Alice,” then Alice’s message is secure, albeit only accidentally so. The presence of names under both crypto layers is crucial, but including both names is not strictly necessary:

1. If Alice wants to use Sign & Encrypt, then she needs to enclose only Bob’s name, because this will link the outer layer’s key to the inner layer.

$$A \rightarrow B : \{\{Bob, msg\}^a\}^B \quad (21)$$

By signing Bob’s name into her message, Alice explicitly identifies him as her intended recipient. This is equivalent to I’Anson’s repair for X.509v1, as discussed above in Section 3.1.

2. If Alice prefers instead to use Encrypt & Sign, then she should encrypt her own name along with her message, and should sign her message-plaintext outside the ciphertext, so as to block Anderson’s plaintext-replacement attack:

$$A \rightarrow B : \{\{Alice, msg\}^B, \#msg\}^a \quad (22)$$

Again, this links the outer layer’s key-pair to the inner layer, and prevents an attacker from replacing Alice’s signature. Encrypting the sender’s name works in a subtle way to prove that Alice performed the encryption: The enclosed name shows that the encryptor intends for the outer signature to carry the same name (Alice’s). The outer signature, in turn, says that Alice did indeed touch the ciphertext. Therefore, Bob knows that Alice performed the encryption.

3. If Alice encloses *both names* in the message-body, she can avoid having to pay attention to cryptographic choices early on, while she’s formatting her message text. She can send to Bob in either of two ways:

$$\begin{aligned} A \rightarrow B &: \{\{“A \rightarrow B”, msg\}^B, \#msg\}^a \\ A \rightarrow B &: \{\{“A \rightarrow B”, msg\}^a\}^B \end{aligned} \quad (23)$$

These two-name formats might be suitable for a flexible standards-specification like S/MIME, in which the layers of crypto can be applied in any order. Always enclosing both names with the message is simpler than judging on the fly which names to enclose, depending on the choice of cryptographic wrappings.

These repairs are rational examples of Martín Abadi’s and Catherine Meadows’ rule-of-thumb for designing security protocols:

- Abadi: “If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal’s name explicitly in the message.” (Principle 3 in [1])
- Meadows: “In general, it’s safer to include names explicitly inside crypto protocols’ messages.” [18]

5.2 Sign/Encrypt/Sign

Surprisingly, we can get an effective repair for S&E, if Alice signs and encrypts the plaintext, and then she signs the ciphertext, too:⁸

$$A \rightarrow B : \{\{\{msg\}^a\}^B, \#B\}^a \quad (24)$$

(Here, $\#B$ means Alice hashes Bob’s key, not his name.) This message means:

- Inner Signature: “Alice wrote the plaintext;”
- Encryption: “Only Bob can see the plaintext;”
- Outer Signature: “Alice used key B to encrypt.”

Bob can conclude not only that Alice wrote the message, but that she also encrypted it. Seen another way, S/E/S is a variation on including the sender’s name inside the plaintext, which then is encrypted and signed (see Sec. 5.1, bullet 2). The inner signature’s key links the encryption-layer to the outer signature’s layer. Alice signs Bob’s key, so as to protect herself from Anderson’s plaintext-substitution attack.

⁸For notational simplicity, we represent these signatures as $\{stuff\}^a$, instead of as $stuff, \#\{stuff\}^a$.

5.3 Encrypt/Sign/Encrypt

Conversely, Alice can get the same security guarantees by re-encrypting her ciphertext's signature:

$$A \rightarrow B : \{\{\{msg\}^B, \#msg\}^a\}^B \quad (25)$$

This message means:

- First Encryption: “Only Bob sees the plaintext;”
- Signature: “Alice wrote the plaintext and the ciphertext;”
- Outer Encryption: “Only Bob can see that Alice wrote the plaintext and ciphertext.”

Bob cannot forward the message without invalidating Alice’s signature. The outer encryption serves to prevent an attacker from replacing Alice’s signature. As with S/E/S, E/S/E is a variant of including the recipient’s name inside the plaintext, which is then signed and encrypted (see Sec. 5.1, bullet 1). Alice signs her plaintext along with her ciphertext [27], so as to protect herself from Anderson’s plaintext-substitution attack. At the same time, Alice’s signed plaintext gives Bob non-repudiation.

5.4 Costs and Advantages

Of course, the naming repairs and the double-signed repairs offer different trade-offs. The naming repairs bring no performance cost, but they do require new standards, and those standards would arguably be more intricate than the current standards (because interdependence of layers conflicts with arbitrary nesting of layers). The double-signed repairs are quite expensive in speed, but they have two virtues:

- Double-signing is quite compatible with the existing CMS and S/MIME specifications. The only change double-signing would bring is that the standard would have to require that the recipient check the innermost layer’s key against the outermost layer’s key.
- For some applications, double-signing may be preferable to having to put names into message-bodies or payloads.

Overall, it’s clear that the simplest repair is to add the recipient’s name, then Sign & Encrypt(§ 5.1, bullets 1 and 3). The other solutions all require an extra hash of the message or of the encrypting key, so as to block Anderson’s plaintext-replacement attack.

6 Conclusions

We have presented a forensic history of how naïve Sign & Encrypt, an insecure cryptographic primitive, has come to be widely trusted, standardized, and implemented, despite its insecurity. The notion that naïve Sign & Encrypt is secure seems to have arisen with PGP’s first description in 1986. This mistake was reinforced by a misstatement in a paper that proposed several repairs for X.509v1. Since then, all of the leading standards for file-encryption and for secure e-mail have relied on naïve Sign & Encrypt. Some of these defective standards can be fixed easily, but for others, the repair would become intricate. Secure-session protocols and authentication protocols typically do not rely on naïve Sign & Encrypt, so they are not affected by this paper’s findings.

The weakness of naïve Sign & Encrypt is somewhat subtle, but it is easily fixed in several ways. The repairs all show that Signing and Encryption should not be viewed as independent operations; the repairs presented here all rely on linking the outer operation’s key to the inner operation’s payload. This realization, that public-key operations are not necessarily so independent as they’re commonly thought to be, and that coupling two layers together is a profitable primitive, may prove to be a novel and useful axiom for beginning protocol designers and analysts.

7 Acknowledgements

I have had profitable discussions about these ideas with many expert critics: Martin Abadi, Ross Anderson, Marc Branchaud, Dave Carver, Dan Geer, Peter Gutmann, Philip Hallam-Baker, Paul Hoffman, Russ Housley, Steve Kent, Norbert Leser, John Linn, Ellen McDermott, Joseph Reagle, Ed Simon, Win Treese, Charlie Reitzel, Ralph Swick, and Henry Tumblin. I thank all of these people for their patient attention.

References

- [1] M. Abadi and R. Needham, “Prudent Engineering Practice for Cryptographic Protocols,” *Digital SRC Research Report #125* (June 1, 1994).
- [2] R. Anderson and R. Needham, “Robustness Principles for Public Key Protocols,” in *Lecture Notes in Computer Science 963*, Don Coppersmith (Ed.), *Advances in Cryptology - CRYPTO ’95*, pp. 236-247. Springer-Verlag, 1995.

- [3] S. Crocker, N. Freed, J. Galvin, S. Murphy, Internet RFC 1848 "MIME Object Security Services," October 1995.
<http://www.faqs.org/rfcs/rfc1848.html>
 - ftp://ftp.isi.edu/in-notes/rfc{989,1040}.txt
 - ftp://ftp.isi.edu/in-notes/rfc{1113,1421}.txt
- [4] M. Burrows, M. Abadi, and R. Needham, "A Logic of Authentication," *Proc. R. Soc. Lond. A* 426(1989) pp. 233-271.
- [5] T. Dierks and C. Allen, Internet RFC 2246 "The TLS Protocol Version 1.0," January 1999.
<ftp://ftp.isi.edu/in-notes/rfc2246.txt>
- [6] D. Eastlake, J. Reagle, and D. Solo (Editors), "XML-Signature Syntax and Processing: W3C Working Draft 18-September-2000," <http://www.w3.org/TR/xmlsig-core/>
- [7] P. Hoffman, Internet RFC 2634 "Enhanced Security Services for S/MIME," June 1999.
<ftp://ftp.isi.edu/in-notes/rfc2634.txt>
- [8] Paul Hoffman, personal communication.
- [9] R. Housley, Internet RFC 2630 "Cryptographic Message Syntax," June 1999.
<ftp://ftp.isi.edu/in-notes/rfc2630.txt>
- [10] Russ Housley, personal communication.
- [11] C. I'Anson and C. Mitchell, "Security Defects in CCITT Recommendation X.509 - The Directory Authentication Framework," *ACM Comp. Comm. Rev.*, (Apr '90), pp. 30-34.
- [12] B. Fraser, T.Y. Ts'o, J. Schiller, M. Leech, "IP Security Protocol (IPSEC) Charter," <http://www.ietf.org/html.charters/ipsec-charter.html>
- [13] A. Joux and R. Lercier, "Discrete logarithms in GF(p)", April 17 2001. Announcement on the Number Theory Mailing List NMBRTHRY. <http://www.medicis.polytechnique.fr/~lercier/talk/ecc99.ps.gz>.
- [14] International Telegraph and Telephone Consultative Committee (CCITT). Recommendation X.509: The Directory - Authentication Framework. In *Data Communications Network Directory, Recommendations X.500-X.521*, pp. 48-81. Vol. 8, Fascicle 8.8 of *CCITT Blue Book*. Geneva: International Telecommunication Union, 1989.
- [15] J. Linn, Internet RFCs 989, 1040, 1113, 1421, "Privacy Enhancements for Internet Electronic Mail: Part 1: Message Encryption and Authentication Procedures," February 1987, January 1988, August 1989, February 1993.
- [16] John Linn, personal communication.
- [17] Stephen Kent, personal communication.
- [18] C. Meadows, "Verification of Security Protocols," lecture at 1996 RSA Cryptographers' Colloquium, Palo Alto, CA.
- [19] C. Neuman and J. Kohl, Internet RFC 1510 "The Kerberos Network Authentication Service (V5)", September 1993. <ftp://ftp.isi.edu/in-notes/rfc1510.txt>
- [20] B. Ramsdell, Internet RFC 2633 "S/MIME Version 3 Message Specification," June 1999. <ftp://ftp.isi.edu/in-notes/rfc2633.txt>
- [21] R. Rivest, A. Shamir, L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Comm. ACM*, v. 21, 2, Feb. '78, pp. 120-126.
- [22] RSA Laboratories, "PKCS#1: RSA Cryptography Standard," version 2.0. Amendment 1: "Multi-Prime RSA." <http://www.rsasecurity.com/rsalabs/pkcs/pkcs-1/>
- [23] RSA Laboratories, "PKCS#7: Cryptographic Message Syntax Standard," Version 12.5, Nov. 1, 1993. <http://www.rsasecurity.com/rsalabs/pkcs/pkcs-7/>
- [24] RSA Laboratories, "PKCS#9 v2.0: Selected Object Classes and Attribute Types," February 25, 2000. <http://www.rsasecurity.com/rsalabs/pkcs/pkcs-9/>
- [25] Joseph Reagle, personal communication.
- [26] J. Reagle, "XML Encryption Requirements," W3C Working Draft 2001-April-20, <http://www.w3.org/TR/2001/WD-xmlencryption-req-20010420>.
- [27] Ed Simon, personal communication.
- [28] Visa International and MasterCard, "Secure Electronic Transactions Protocol Specification," <http://www.setco.org/set.specifications.html>.
- [29] P. Zimmermann, "The Official PGP User's Guide," MIT Press (1995).
- [30] P. Zimmermann, "A Proposed Standard Format for RSA Cryptosystems," *IEEE Computer* 19(9): 21-34 (1986).

Unifying File System Protection

Christopher A. Stein¹, John H. Howard², Margo I. Seltzer¹

¹ *Harvard University*

² *Sun Microsystems*

Abstract

This paper describes an efficient and elegant architecture for unifying the meta-data protection of journaling file systems with the data integrity protection of collision-resistant cryptographic hashes. Traditional file system journaling protects the ordering of meta-data operations to maintain consistency in the presence of crashes. However, journaling does not protect important system meta-data and application data from modification or misrepresentation by faulty or malicious storage devices. With the introduction of both storage-area networking and increasingly complex storage systems into server architectures, these threats become an important concern.

This paper presents the protected file system (PFS), a file system that unifies the meta-data update protection of journaling with strong data integrity. PFS computes hashes from file system blocks and uses these hashes to later verify the correctness of their contents. Hashes are stored within a system log, apart from the blocks they describe, but potentially on the same storage system. The write-ahead logging (WAL) protocol and the file system buffer cache are used to aggregate hash writes and allow hash computations and writes to proceed in the background.

PFS does not require the sharing of secrets between the operating system and the storage system nor the deployment of any special cryptographic firmware or hardware. PFS is an end-to-end solution and will work with any block-oriented device, from a disk drive to a monolithic RAID system, without modification.

1 Introduction

For a variety of economic and management reasons, server operating systems are moving towards a looser coupling with their storage subsystems [6]. There are two important components to this change. First, the technology connecting the server operating system to the underlying storage system is changing. Busses are

being replaced by networking fabrics known as storage area networks (SAN) that utilize protocols such as Fibre Channel or Gigabit Ethernet [2]. In the future we may even see IP networks within the SAN. The IETF IP Storage working group is designing a protocol for transporting block-level storage commands over IP networks [10]. This shift from protected busses to networking fabrics introduces an increased risk of malicious intrusion.

Storage Service Providers (SSPs) offer storage outsourcing services, connecting a client's application server to their storage system, which can be partitioned across several clients. In these cases, strong protection guarantees become important because the network between the server operating system and the storage system now spans commercial boundaries. The server and the storage are no longer under a single administrative control. The client will undoubtedly have a data integrity contract with the SSP, but an alarm should be raised immediately if the SSP, either intentionally or unintentionally, delivers bad data.

Second, the complexity of storage is increasing. Storage systems have evolved from direct attached hard disks to large network-attached disk arrays with internal RAIDs. These systems have complicated firmware, and many run non-trivial operating systems. This new complexity introduces software failure modes that are not described by the traditional fail-stop model of hardware.

Many problematic scenarios can arise in this new environment. For example, malicious intruders spoof blocks to the server, a software bug on the storage corrupts data, or the storage receives or transmits a different block than the one requested. The server operating system must protect itself against all such failures. Blocks containing file system meta-data are particularly important because they contain operating system state and will crash the server operating system, and consequently all applications, if they are corrupted.

This paper introduces the Protected File System (PFS). PFS is intended for server operating systems and provides strong integrity at the level of file system blocks. Block hashes are computed with a collision-resistant hash function and are subsequently used to verify block reads from the storage. PFS does not change the file system interface. Applications are free to open, close, read and write files as they always have. PFS also makes no changes to the on-disk layout of data and meta-data. This allows for backward compatibility with existing data and the freedom to use existing utilities. PFS unifies block verification with the existing file system protection mechanism, the file system journal. This unification is achieved using a generic system service known as the write-ahead file system (WAFS) [19]. The WAFS stores records and provides its clients with log sequence numbers (LSNs) for their record writes. PFS uses the WAFS for both journaling and hash logging. This centralizes recovery in a single component, consolidates potentially disparate I/O streams into a single sequential stream, and avoids code duplication.

PFS uses collision-resistant hashes to protect data integrity. Collision-resistant hashes have the property that it is computationally infeasible to find two different data blocks with the same hash value. SHA-1 [4] is one popular such hash, mapping a data block of any size to a 20-byte hash value. MD-5 [17] is another, mapping to a 16-byte value.

Section 2 outlines the properties that we sought in a solution and discusses some candidate architectures. The goal is to give readers an understanding of the process we followed in order to arrive at the PFS architecture. Section 3 discusses related work. Section 4 provides an overview of the PFS architecture, and Section 5 discusses some of the more important issues in detail. Section 6 presents our performance results, and Section 7 concludes.

2 Desired Properties and Potential Solutions

At the beginning of this project, we set out the following requirements for our architecture.

- Detect all bad blocks. With a very high degree of certainty, any unauthorized block modification must be detected by the file system. This includes both data and meta-data blocks. With the same degree of certainty, the file system must detect that the storage returns a block other than the one requested.
- No changes to storage systems. No special cryptographic logic, either hardware or software, should

be necessary. File system block sizes must remain a power of two so that they can be expressed efficiently in terms of storage blocks (e.g., sectors), which tend to default to a power of two smaller than traditional file system blocks.

- Straightforward to implement in both legacy and modern file systems. The on-disk structure of the file system must be maintained. The architecture must interact well with the operating system buffer cache. The popular file system interface must remain unchanged.
- Minimal requirements for local non-volatile storage. Centralizing non-volatile state on the storage has the added benefit that a failed server can be quickly replaced by another trusted server.
- High performance. Sacrifice as little performance as possible.

A first potential solution would be to compute a hash for each block and store it along with the block, either appended to it or in a special header. When the data is faulted in, the operating system computes the hash and compares it with the hash value stored alongside the block. There are several problems with this solution. First, the storage system block size must be larger than the file system block size by the size of the hash, which is 20 bytes under SHA-1 and 16 bytes for MD-5. This may rule out a large class of storage devices that lack flexible block sizes. Second, malicious intruders with knowledge of the hash function can spoof data and generate a hash, which will be verified successfully, but for incorrect data. Third, a class of storage failures are not protected against. For example, the operating system requests block X and the storage system returns block Y along with the hash appended to Y, the operating system will compute the hash from the value of Y and compare with the appended one. The two, assuming no other failure, will be the same, and the operating system will incorrectly conclude that the block is correct. Simple self-certifying blocks can solve the third of these concerns. A self-certifying block combines both the hash of the data and the block number, so if a block other than the one requested is delivered, that fact will be detected. Signed self-certifying blocks can solve the second and third of these concerns, using a private key to generate a signature so that malicious intruders cannot spoof blocks with correct hashes. However, neither simple nor signed self-certifying blocks can solve the first concern. Under both the simple and signed self-certifying block potential solutions, the storage block size must be larger than the file system block size to accommodate the block number or the hash.

A second potential solution stores the hashes with the file system pointers. For example, the UFS inode, the on-disk structure representing a file, contains an array of pointers to data blocks and, for larger files, pointers to blocks containing pointers. Data block hashes are stored alongside the pointers, within the meta-data. This solution separates the storage of hashes from their data blocks, overcoming the need to increase the storage system block size. Unfortunately, this solution has several problems of its own. First, new dependencies are introduced between file data and meta-data. Suppose the block's hash was written alongside the inode's block pointer and before the block write. If the system crashed before the modified block reached storage, the hash and block would be inconsistent after a reboot. The hash and block would also be inconsistent if the hash was to be written after the block, but the system crashed before the hash could be committed. The update of the block and hash needs to be atomic or at least subject to strict write ordering constraints. Unfortunately, while it might be possible to overcome the ordering challenges associated with this potential solution by applying sophisticated design, a more serious problem exists.

Storing the hashes with the file system pointers presents another problem that is more serious. Many pieces of meta-data are indexed directly and not referenced via pointers. These include the blocks containing inodes, cylinder groups, and superblocks. User data, if corrupted, will not crash the system because it is opaque and passed through to applications. Meta-data, on the other hand, is interpreted by the operating system and its corruption can crash the system. The protection of such meta-data is critical, because these are the blocks whose correctness is most important for system stability. If the inodes are not protected from modification, then a malicious entity could change the hash value associated with a block pointer to match that of a spoofed block. Again, a signature scheme could be used to sign meta-data. These techniques could be feasible under read-only workloads, but become problematic under workloads with frequently changing meta-data, due to the introduction of update dependencies and the need to sign meta-data.

The best features of the above-mentioned two solutions—hash protection of all blocks, both data and meta-data, and the separation of hashes from their corresponding blocks—are both achieved through a technique that we have developed called *hash logging*. Hashes, along with a unique block identifier, are written as records into a log, a separate append-only system file. The hashes of all blocks, both data and meta-data, are stored within this log and are thereby separated from the

blocks they describe. Log records are keyed by monotonically increasing LSNs. When record writes reach the end of the log, they start anew at the beginning (this necessitates a log space reclamation mechanism, which will be described in section 5).

3 Related Work

Fu et al. describe the read-only secure file system (SFSRO) [5]. SFSRO is most like the second potential solution described above. File system blocks are named and requested by their hash value. The traditional disk address pointers contained in file system index nodes are replaced by hash values. This scheme depends on the collision-resistant property of cryptographic hashes. As SFSRO is designed for read-only workloads it does not handle frequent updates well. If the contents of the file system change, a database must be reconstructed on a trusted server and shipped to the untrusted server, where it cannot be modified. In contrast, PFS allows for modification directly on the untrusted storage server. PFS protects data and meta-data at the level of blocks, including all the blocks required to implement a file system on storage: data blocks, allocation bitmaps, inodes, and superblocks. SFSRO protects data blocks and inodes, but not other meta-data.

The Trusted Database (TDB) implements a database buffer cache on an untrusted store [14]. TDB shares several characteristics in common with PFS and SFSRO. All three use collision-resistant hashes to verify blocks. TDB uses a log-structured store for both block and hash writes. PFS and TDB have different philosophies, however. TDB is a monolithic database system that presents a new interface and a new on-disk storage format. In contrast, PFS is an existing file system modified internally to do hash data protection. The logic fits within the prior file system architecture — the most interesting characteristic being the unification of hash and meta-data logging within the file system journal. The popular and well-documented file system interface remains unchanged so that applications can benefit from PFS protection transparently. Also, the on-disk format of the file system remains unchanged, so that large file system partitions run under PFS immediately without copyovers and utilities that access the raw disk interface continue to work (e.g., dump, restore, fsck).

Tripwire is a user-level system administration tool that computes hashes on a per-file basis and stores these in a protected database [12]. To verify and check for intrusion, Tripwire computes the hashes of specified files and compares these against the database, raising a flag for the system administrator if the two differ. Tripwire will

not update a file's hash value when the file is modified by a trusted party. Tripwire simply identifies changed files. Figuring out if a file was changed by a malicious user or a trusted user is a problem left to the system administrator. PFS protects the system at a lower level of abstraction; the file system block, protecting file system meta-data in addition to the file data. PFS block updates will never be committed to storage without the hash being computed and committed beforehand.

For high performance, PFS commits block hashes and meta-data update (journal) records to the same log. The idea of shared logging was investigated within the Quicksilver operating system project [9]. Quicksilver was a distributed microkernel operating system built at IBM research in the mid-1980s for System/6000 workstations. The salient feature of Quicksilver was its use of transactions for recovery, failure notification, and resource reclamation. As a microkernel, Quicksilver typically implemented system services in user-level servers. The log manager server was used primarily by the transaction manager, but was theoretically available

to other subsystems. Later work explored sharing the log service across multiple resources [3]. The WAFS used by PFS is also a service for shared logging, built into a monolithic kernel architecture. Other work has explored sharing the WAFS service between the kernel and user applications [20].

4 PFS Architecture Overview

4.1 Block Modification and Writes

When buffered file system blocks are modified, either through direct user write system calls or internally generated meta-data updates, PFS flags the buffers to indicate that their hashes must be computed and logged before they may be written to the storage system. A background system thread known as *hashd* wakes periodically and cycles through the dirty buffers, identifying those buffers for which a hash must be computed. Once such a buffer is identified, *hashd* applies the hash function to the buffer's data block, yielding a hash value that is packed into a record along with the block identifier and written to the log. The log write returns an LSN.

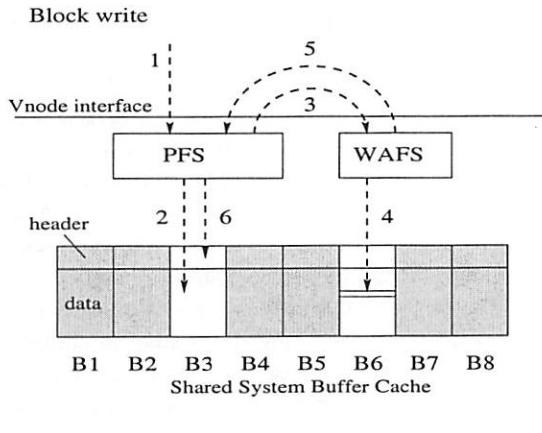


Figure 1: Block Write Example. (1) Data is written by an application. This step will not occur if the block modification is an internally generated meta-data update. (2) PFS writes data into block buffer B3 currently resident in the buffer cache. A buffer flag is set to indicate that the hash must be computed. At this point, the system is free to return. Steps 3-6 happen asynchronously within the context of the background process *hashd*. (3) PFS applies a hash function to the data contents of B3. This hash and B3's physical block number are encapsulated within a record and written to the WAFS via the vnode interface. (4) The WAFS takes the record and does a buffered write to the head of its log in B6. (5) It returns the LSN. (6) PFS stores the LSN in B3's buffer header.

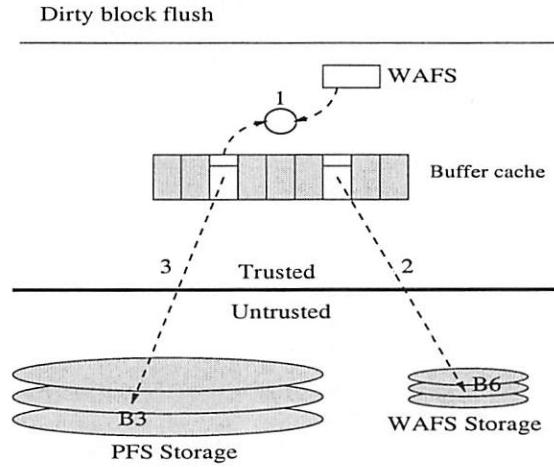


Figure 2: Dirty Block Flush Example. (1) This is the write-ahead comparison. The LSN stored in buffer B3 is compared to the WAFS highest stable LSN. If the buffer's LSN is less than or equal to the highest stable LSN, then the log record describing B3 must be stable. It is safe to write B3 without forcing the log (go to Step 3). (2) The hash record describing B3 needs to be committed. Flush the log up to the LSN stored in B3's buffer. The record happens to be stored within B6, so B6 is written. (3) Write block B3.

This LSN is stored in the block's buffer. These steps are shown in Figure 1. Steps 1 and 2 of that figure occur synchronously with the system call. Steps 3 through 6 take place in the background in the context of hashd. None of these steps require communication with the storage system.

To ensure the integrity of the data, PFS uses the well-known write-ahead logging protocol (WAL) [7]. Before PFS finally decides to write a buffer, it must ensure that the corresponding hash record has reached stable storage. If the block were written otherwise and the system crashed, a subsequent comparison of the old hash against the new data would erroneously conclude that the data was bad. Therefore, hashes must reach stable storage before their corresponding blocks. LSNs are used to enforce this dependency.

Before a block is written to disk, the LSN stored within its buffer header is compared with the stable LSN of the log. If it is greater than the stable LSN, then in order for this write to proceed, the log must first be flushed up to the buffer's LSN. This is shown in Figure 2.

Using an asynchronous thread to compute and log the hashes has several important advantages. First, it removes the hash computation from the system-call path, reducing the likelihood of needless hash recomputations. Second, it increases the likelihood that a hash will be stable by the time its corresponding dirty block reaches the device driver. If the hash were not stable by this time, two I/Os would be necessary. A synchronous write of the block's hash to the log followed by the data block write. Third, using an asynchronous thread allows hash records to be batched together. As long as WAL is followed, integrity will be protected and the hash record can be written asynchronously.

4.2 Reading a Block

The *block map* is the PFS data structure used to verify block reads. It contains the committed hash for every block, indexed by file system block number. The block map resides in kernel virtual memory and need not be entirely resident in physical memory. The portion of the map present in memory is a function of file system locality. Using paging to cache the recently accessed pages of the block map requires a trusted swap device. The block map is a volatile data structure. When the system crashes, it is reconstructed from the hash log records contained within the WAWS.

On a read, if PFS does not find a block in the buffer cache, it issues a read request to the storage system.

Upon receiving the block from storage, PFS computes the hash and compares this value with the value in the block map. If the block fails this verification step, it is expelled and the associated system call fails.

4.3 Journaling and Hash Logging

File systems that do not employ techniques such as journaling or soft updates do not perform well under meta-data intensive workloads [19]. This is because they must perform synchronous writes to order meta-data updates. Journaling solves this problem by allowing the file system to buffer meta-data updates, using a log and WAL to enforce dependencies, much as hash logging does. Journaling can be thought of as protecting the operating system from itself or, at a finer level of detail, protecting the file system from the buffer cache, which is responsible for committing delayed writes, but has no knowledge of crucial file-system-specific update dependencies. Hash logging benefits from the delayed meta-data updates of journaling. If Step 2 of Figure 1 were a synchronous meta-data update, then all of the following steps within Figure 1 and the two writes of Figure 2 would have to be performed synchronously. By allowing meta-data updates to be cached delayed-write, journaling gives the system the freedom to perform hash computation in the background and to batch hash record writes.

Rather than have two logs in the system, each serving its own LSNs to enforce WAL, it makes sense to unify the two logs into one single system WAL service. PFS uses the *write-ahead file system* (WAWS) [19] for this purpose, employing it for both meta-data and hash logging. The WAWS resides on a separate partition from PFS. This allows the WAWS to be located on a different storage system from PFS, potentially removing disk head contention. On systems with NVRAM, the WAWS could be placed there.

The WAWS is not confined to trusted local storage. Therefore, it must be protected from the same classes of faults and attacks as PFS. If an adversary were able to spoof WAWS blocks, then PFS would no longer be able to protect its own blocks. During a server crash, an adversary could read the log records, which are stored in the clear on the storage, find the most recent record describing a PFS block, modify the corresponding PFS block, compute the new hash and insert it in the WAWS block in place of the old value. From this example, it is clear that the protection of WAWS blocks is paramount to the protection of PFS blocks.

WAWS does not rely on an external mechanism for its protection, instead storing an *authentication tag* in the

header of every WAFS block. The authentication tag is generated by a message authentication code (MAC). Unlike hash functions, MACs are parameterized by a secret key. Recent work has shown that MACs can be constructed from hash functions with a degree of security that is provably strong [1]. The benefit of this approach is performance, which is essentially that of the underlying hash function.

Unlike PFS, the WAFS is able to include the tag within the file system block because its units of read/write access are variable length records. Any notion of file system blocks is withheld from its clients, which read and write records, not blocks. The WAFS header also includes an LSN. This is protected by the authentication tag and used to verify that the correct block was returned. Since the server is the only system that requires verification of WAFS blocks, public key technology need not be used.

The authentication tag is computed when the WAFS block is written to storage. Since WAFS blocks are neither read nor rewritten during normal operation, the

authentication tag will only need to be recomputed for partial block writes that are forced to storage.

The secret key used for HMAC computation must be secure and protected on a trusted device. It could be located on the server's trusted disk and managed as a dynamically loadable kernel module. More advanced technology such as smart cards could be used.

Figure 3 shows the journal and hash log multiplexed on the WAFS.

5 PFS Architecture Details

5.1 Asynchronous I/Os

After the server sends the write request to the storage and before it receives the I/O completion notification, the block may be either in the new or old state. If the server crashes before receiving notification, then it must have both hashes stable for comparison in the next session. If the write did not make it to the storage system, the block is still correct so the old hash must be available for comparison. The block map contains the committed hash of every block. Hashes are only entered into

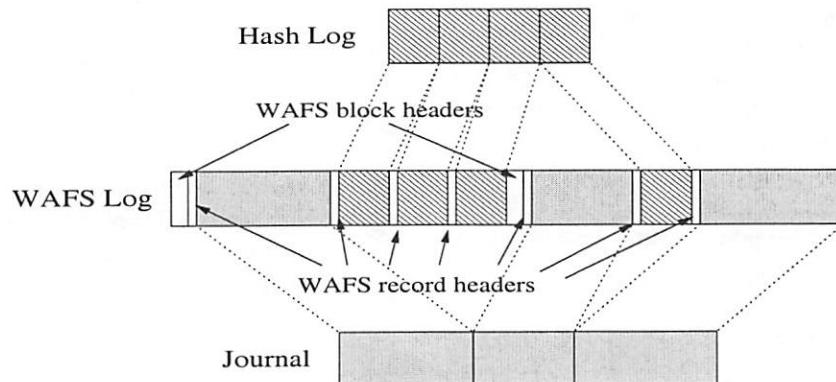


Figure 3: Multiplexing the WAFS. This figure shows two physical WAFS blocks containing four hash log records (diagonal-striped blocks) and 3 journal records (grey blocks). PFS uses the WAFS for its hash logging and journaling. The journaling and hash logging components write records during normal operation. Each component sees a distinct logical log, but the records are physically interleaved. Records are read only during recovery. WAFS does not export a block abstraction and hides the WAFS block header from its clients. The header contains an LSN needed during recovery and an authentication tag to verify WAFS block reads. The WAFS prepends a record header to every record write. This contains a client identifier used by the cursor read routines to ensure that the block map recovery code does not retrieve journal records and vice versa.

the block map once the I/O completion interrupt has been received. The *async map* is a small hash table containing all hashes describing block updates sent to storage, but for which PFS has not received I/O completion notification. Upon receiving this notification, the hash is removed from the *async map* and inserted into the block map, overwriting the previous value.

5.2 Checkpointing

For a 16GB partition with 8KB file system blocks, the block map will have 2M entries. If the hashes are MD-5, the block map will be 32MB. Clearly, it is undesirable to checkpoint the entire block map at once. The block I/O interrupt code accesses both the block map and the *async map*, so access to these data structures must be exclusively locked during the checkpoint. Locking out block interrupts in order to checkpoint tens of megabytes of data would undoubtedly result in dropped interrupts. In addition, system call latencies would be severely skewed and unpredictable. We solve this problem with partial checkpoints. The block map is broken up into distinct *chunks* approximately the size of the data portion of WAIFS file system blocks. Each chunk is checkpointed independently and the chunks are selected in a round-robin fashion.

A chunk checkpoint is not complete until a small record known as the *async record* has been committed to the log following the chunk. The *async record* contains the contents of the *async map* for the file system blocks described by the chunk. These are the block I/Os in progress at the time of the checkpoint. Since there can be only one outstanding I/O on any particular block, the current state of the block is either described by the hash in the chunk or the hash in the *async record*. Once a chunk checkpoint completes, the log is free to reclaim earlier records for blocks described by the chunk.

For a given block, many hash records may be written after the chunk checkpoint. PFS must attempt to bound the number of potentially valid hashes, or *candidate hashes*, for a given block. Having many candidate hashes to test would slow down recovery and adversely affect security. One way to bound the number of candidate hashes per block would be to insert code in the I/O completion interrupt handler to write a special record to the hash log. This record would contain the hash value and block number like other records, but its type field would identify it as an I/O completion record. During recovery, on a read of such a record, the recovery code would deprecate all earlier hash values associated with the block. This solution would place a fairly tight bound on the number of candidate hashes per block, dependent

on the flush rate of the buffer cache and the hash computation rate of hashd. The bound would be inversely proportional to the flush rate of the buffer cache and directly proportional to the buffer processing rate of hashd. Unfortunately, there is a fundamental architectural problem. Locks (e.g., the WAIFS vnode lock) would have to be acquired during interrupt processing, which runs at a higher priority level than the top half of the kernel. This would introduce the potential for deadlock.

Our solution places the bounding responsibility within hashd. In order to bound the number of candidate hashes, every PFS buffer contains two LSNs. The first is the LSN immediately following the buffer's most recently written log record. The hash stored in the record preceding this LSN might not describe a committed PFS block. This LSN is known as the *buffer-end*. The second is the LSN immediately following the most recent hash describing a stable version of this buffer. This is known as the *buffer-begin*. For a given buffer, all of the hashes between buffer-begin and buffer-end have not had their corresponding data committed to disk. As hashd computes and logs hash records, buffers' buffer-end LSNs will move forward. When the PFS block is written to disk, the I/O completion interrupt will copy buffer-end into buffer-begin, moving it forward. At this point, the most recent hash in the log must be stable, due to the protection of the WAL protocol, and it must describe this particular write because the buffer is locked during physical I/O. This process is shown in Figure 4.

When hashd executes, it scans the buffer-begin values of all the PFS buffers, selecting the minimum from the buffers for which buffer-begin differs from buffer-end. This LSN is written to the log in a special record. This LSN has the property that if there are no hash records describing a particular block after this LSN, then the most recent hash record preceding the LSN describes this block. In Section 5.3 we will describe how this LSN, known as *logged*, is used in recovery.

The WAIFS is a finite circular log and is responsible for managing this space. When the WAIFS reaches a minimum free space threshold, it synchronously checkpoints its clients. The journal follows the methodology described in Seltzer et al. [19] and is beyond the scope of this paper. WAIFS checkpoints the hash log by calling a PFS handler and providing a minimum acceptable LSN. The handler compares this against the LSN of the oldest live checkpoint chunk. If it is less, then PFS simply returns the LSN of the oldest checkpoint without any I/O. Otherwise, PFS checkpoints enough chunks for

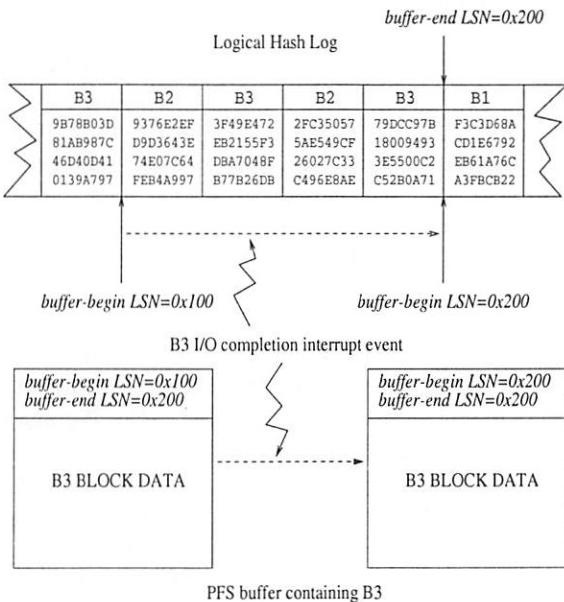


Figure 4: Updating the buffer-begin LSN. On reception of a block I/O completion interrupt, the buffer-end LSN is copied into the buffer-begin LSN. These two LSNs are used by hashd to compute the global logged LSN. If the buffer-begin and buffer-end LSNs are equal, the block contained within the buffer does not have any logged hashes that must be tested and resolved during recovery.

the oldest live LSN to move beyond the minimum acceptable LSN. This illustrates the fact that the WAFS space reclamation checkpoint must occur with enough free space to accomodate possible hash log checkpoints. Journal checkpointing will also require free space. Interestingly, the minimum free space is easier to bound for hash logging than for journaling. Heuristic bounds for journaling use the maximum number of open file descriptors and the maximum number of journal entries per system call [19]. The upper bound for hash logging is simply the size of the block map plus a small amount of space for an async record.

5.3 Recovery

The WAFS itself must be recovered before any PFS recovery can begin. The most recent checkpoint is found, and the log is then read sequentially until the end of the log is detected. The LSNs stored in the WAFS block headers are monotonically increasing, allowing the recovery code to find the most recent block. The authentication tag protects against incomplete writes.

PFS recovery has two phases: hash recovery and journal recovery. The hash recovery restores the block map to a consistent state. The journal recovery follows and restores the meta-data integrity. The journal recovery

process is unmodified from the original implementation described by Seltzer et al [19].

The hash recovery code reconstructs the block map and verifies the contents of any blocks for which multiple hash values are potentially valid. The hash recovery code receives a log cursor handle from the WAFS that allows it to iterate through the log without receiving the journal records. The hash recovery code finds the oldest valid chunk checkpoint and rolls forward searching for the most recent record containing a logged LSN (see section 5.2 for a description of this special LSN). Once this is found, hash recovery returns to the start of the log for a second pass. Block map recovery requires little I/O up to the logged LSN. This is because every hash value either describes a committed block or is deprecated by a later record that does. Therefore, the hash values can be inserted directly into the block map with no I/O. Once recovery passes the logged LSN, every hash value must be checked for validity. The recovery code maintains a linked list of candidate hashes for every block. On reaching the log end, the hashes are resolved. Every block with multiple valid hashes is read in, the hash function is computed, and the resulting value is compared against all the potentially valid hashes. If it does not match any, recovery fails. Otherwise, the matching hash is inserted into the block map and all the others discarded. This illustrates the importance of bounding the number of candidate hashes per block as described in Section 5.2.

6 Performance Analysis

PFS and WAFS have been implemented in the FreeBSD-4.1 operating system. PFS is a set of changes to the Logging Fast File System (LFFS) [19], a journaling version of FFS [15]. PFS maintains the on-disk structure of FFS, and it uses the WAFS for both its journaling and hash logging as described in this paper.

For these experiments, PFS computes hashes using the MD-5 algorithm and WAFS computes its block authentication tags using the IETF RFC 2104 implementation of HMAC using MD-5 [13]. The HMAC secret key is stored within the kernel binary.

Our test system consists of a single 500Mhz Xeon Pentium III CPU with 512MB RAM and three 9GB 10,000 RPM Seagate Cheetah (ST39102LW) disks. The disks are connected to the host operating system via a single shared Adaptec AHA-2940UW Ultra SCSI card. The first disk contains the operating system and swap space, the second contains a 256MB WAFS partition, and the third contains an 8.5GB test partition.

We ran two macrobenchmarks and a microbenchmark suite. We compare PFS against LFFS. In our experiments, both PFS and LFFS do asynchronous journaling. This means that journal records are not committed to disk before the system call returns. WAL still maintains the ordering of updates so that the file system will be recoverable to a consistent state. However, it is possible that a create will return, the system will crash, and the file will not exist after recovery, violating the durability semantics that FFS has traditionally offered. We believe that these semantics grew out of convenience, rather than application demand. Originally, FFS used synchronous updates to order meta-data operations and ensure recoverability and consistency. Since the updates had to happen anyway, the semantics of durability came at no additional cost. Soft updates [8][16] and asynchronous journaling abandon these semantics. The difference between the performance of synchronous journaling and

either asynchronous journaling or soft updates, whichever is better, is the cost of FFS system call durability. Seltzer et al. determined that the cost of durability far exceeds the cost of integrity [19]. The goal of our benchmarking experiments is to quantify the cost of the block-level integrity protection of PFS.

6.1 Microbenchmarks

The microbenchmarks are similar to those used in recent file system performance studies [18][19]. For file sizes ranging from 16KB to 1MB they create, write, read, then delete a directory hierarchy of files. The file system is unmounted between each operation phase, to ensure that each phase begins with a cold cache. This is useful for isolating the cost of individual operations, but is somewhat artificial. Much of the design of modern file

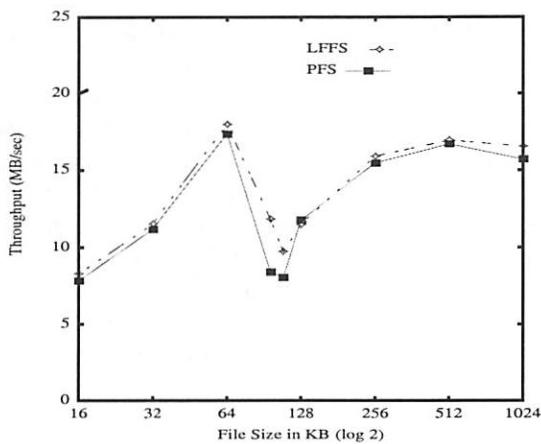


Figure 5: Throughput (MB/s) of creates.

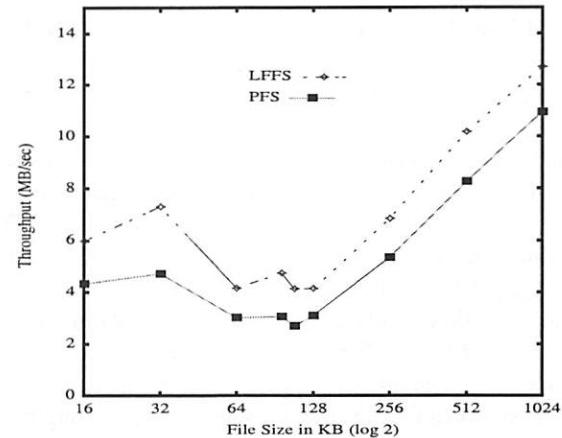


Figure 6: Throughput (MB/s) of writes.

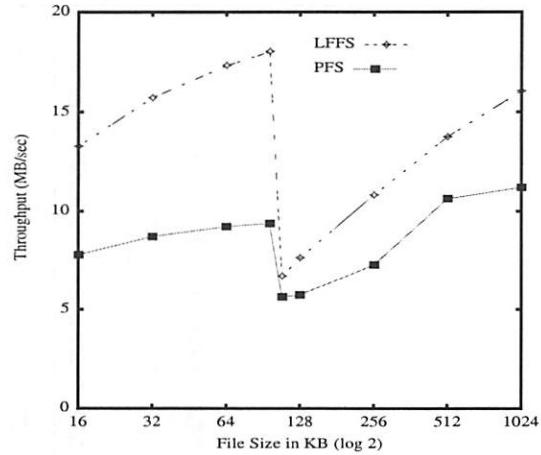


Figure 7: Throughput (MB/s) of reads.

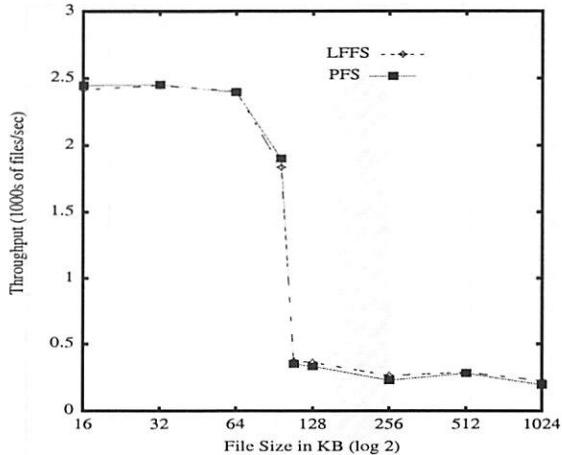


Figure 8: Throughput (thousands of files/s) of deletes.

Figures 5-8: Cold cache microbenchmarks. Figures 5 through 7 show the throughput of both PFS and LFFS expressed in MB per second. Figure 8 shows delete throughput expressed in thousands of files per second. The create and delete benchmarks are meta-data intensive. Note that the scale of the y-axis varies. Each test was run 5 times and standard deviations were small.

systems is predicated on a large cache with a high hit rate.

The hierarchy contains either 128MB of data or 512 files, whichever results in more data. No more than 50 files are allocated per directory to limit pathname lookup times.

The results of the microbenchmarks are shown in Figures 5 through 8. The performance of PFS and LFFS are similar on the create and delete tests. This shows how hash logging benefits from journaling. Journaling allows meta-data updates to be cached and updates combined. The performance of hash logging depends on the ability of the operating system to cache writes. This gives hashd time to generate hash log records and group these records together into full block writes, amortizing the cost of block writes.

On the read and write tests, LFFS outperforms PFS because PFS must compute hashes when blocks are written to and read from disk.

The largest disparity is on the read benchmark. This test has no data locality, but some meta-data locality due to the repeated lookup of directory components. For medium-sized files between 32KB and 96KB LFFS throughput is nearly double that of PFS. At 96KB, the read throughput collapses as the two systems must read the indirect block. For larger files these layout issues, which are independent of the hashing mechanism, play a more significant role and PFS performance improves relative to LFFS. The performance disparity is narrower

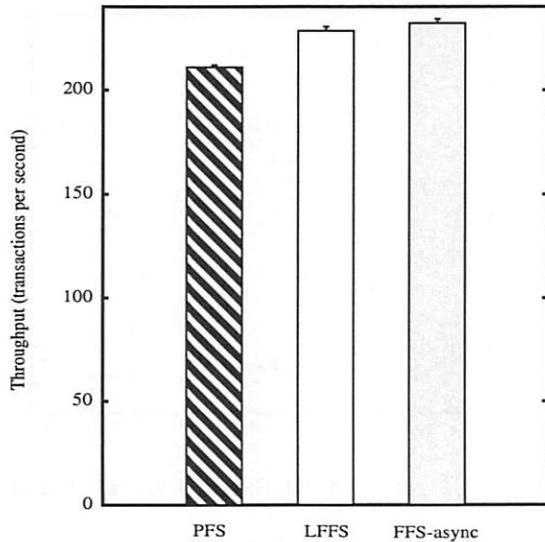


Figure 9: Throughput of PostMark macrobenchmark. Expressed in transactions per second.

on the write test. Here, the buffer cache is able to absorb many of the writes.

6.2 Macrobenchmarks

The microbenchmarks are useful for isolating the operations for which PFS and LFFS performance differs. However, it is difficult to use their results to predict application performance. The goal of our macrobenchmark experiments is to quantify the cost of block-level protection for realistic workloads.

FFS mounted asynchronously (FFS-async) does not make any attempt to ensure file system recoverability or protection. It is useful as an upper bound on performance because it shares layout format and algorithms with PFS and LFFS.

6.2.1 POSTMARK Benchmark

The PostMark benchmark was designed to model a combination of electronic mail, netnews, and e-commerce transactions, the type of load typically seen by Internet Service Providers (ISP) [11]. PostMark is metadata intensive with many small files and high memory pressure. File sizes vary uniformly from 512 bytes to 16KB. Figure 9 shows the performance of the three systems. The throughput of PFS is 7.9% lower than LFFS and 9.1% slower than FFS-async. Although PFS performs well on meta-data operations, due to the fact that it is journaled and meta-data updates are cached, it does not perform comparatively well on small reads and writes.

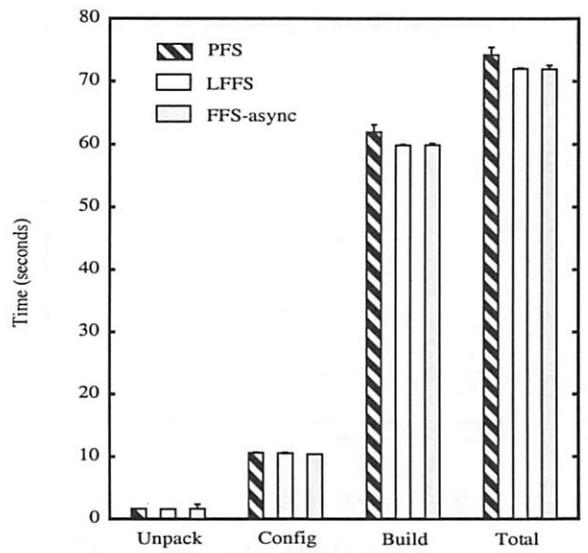


Figure 10: Time (s) taken for SSH-Build to complete. Unlike figures 5 to 9, here lower values indicate higher performance.

6.2.2 SSH-BUILD Benchmark

The SSH-Build benchmark unpacks, configures, and builds the secure shell (SSH) program, a medium sized software package [21]. It consists of three phases. The first, *unpack*, decompresses a tar archive and pulls the source files out of the archive. This phase is relatively short and characterized by meta-data operations. The second, *config*, runs small scripts and programs and generates small files. The third, *build*, compiles the source tree, reading and parsing the source files and generating object files, that are subsequently linked into the executable. Figure 10 shows the performance of the three systems across the three stages. The performance of the three systems on the first two phases is not statistically distinguishable. PFS takes 3.5% longer to complete the build phase. Being less meta-data intensive with larger files than the two earlier phases, this reconciles with our observations from the microbenchmarks.

7 Conclusions

We have presented an evolutionary file system architecture for strong block-level integrity. Our architecture changes neither the file system interface nor its on-disk layout and requires no special support from storage systems. This has been accomplished by modifying the file system to log collision-resistant hashes to a general operating system logging service. This service is shared with the file system journal. For efficiency, a background thread computes and logs hashes, removing the hash computation from the system call path and the log commit from the data I/O path.

We measured the performance of PFS using micro and macrobenchmarks. PFS performance is similar to that of the LFFS journaling file system under the meta-data intensive create and delete tests. Under the read and write tests, PFS incurs hashing costs as data is transmitted to and from storage. On the macrobenchmarks, the overhead of block-level integrity is only 3.5% on SSH-Build and 7.9% on PostMark. We believe this is a small price to pay for strong protection.

Acknowledgements

Thanks to David Molnar for bringing HMAC to our attention. Thanks to David Robinson for comments on an early version of this architecture.

References

- [1] Bellare, M., Canetti, R., Krawczyk, H., "Message Authentication using Hash Functions - The HMAC Construction," *RSA*

Laboratories CryptoBytes, Vol. 2, No. 1, Spring 1996.

- [2] Clark, T., "Designing Storage Area Networks," Addison-Wesley, Reading, MA, 1999.
- [3] Daniels, D., Haskin, R., Reinke, J., Sawdon, W. "Shared Logging Services for Fault-Tolerant Distributed Computing," Position Paper for Fourth ACM SIGOPS European Workshop, Bologna, Italy, Sept. 1990.
- [4] FIPS 180-1. Secure Hash Standard. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.
- [5] Fu, K., Kaashoek, F., Mazieres, D., "Fast and secure distributed read-only file system," *Proceedings of OSDI 2000*, San Diego, CA, October 2000.
- [6] Gibson, G. A., Van Meter, R., "Network Attached Storage Architecture", *Communications of the ACM*, 43(11), November 2000.
- [7] Gray, J., Reuter, A., "Transaction Processing: Concepts and Techniques," Morgan-Kaufmann, 1993.
- [8] Ganger, G. R., Patt, Y. N., "Metadata Update Performance in File Systems," *Proceedings of the First OSDI*, Monterey, CA, Nov. 1994.
- [9] Haskin, R., Malachi, Y., Sawdon, W., Chan, G., "Recovery Management in QuickSilver," *ACM Transactions on Computer Systems*, 6(1), pp. 82-108, Feb. 1988.
- [10] Internet Engineering Task Force, IP Storage (IPS) Working Group Charter, <http://www.ietf.org/html.charters/ips-charter.html> as of March 28, 2001.
- [11] Katcher, J., "PostMark: A New File System Benchmark," Technical Report TR3022, Network Appliance Inc., Oct 1997.
- [12] Kim, G. H., Spafford, E. H., "The design and implementation of Tripwire: A filesystem integrity checker". In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994.
- [13] Krawczyk, H., Bellare, M., Canetti, R., "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Internet Engineering Task Force, Network Research Group, February 1997.
- [14] Maheshwari, U., Vingralek, R., Shapiro, W., "How to Build a Trusted Database System on Untrusted Storage," *Proceedings of OSDI 2000*, San Diego, CA, October 2000.
- [15] McKusick, M.K., Joy, W., Leffler, S., Fabry, R. "A Fast File System for UNIX," *ACM Transactions on Computer Systems* 2(3), pp 181-197. Aug. 1984.
- [16] McKusick, M.K., Ganger, G., "Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem," *Proceedings of*

- the 1999 Freenix track of the USENIX Technical Conference*, pp. 1-17. Jun. 1999.
- [17] Rivest, R., "The MD5 Message-Digest Algorithm," RFC 1321, Internet Engineering Task Force, Network Working Group, April 1992.
 - [18] Seltzer, M. I., Smith, K., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V. "File System Logging versus Clustering: A Performance Comparison," *Proceedings of the 1995 USENIX Technical Conference*, pp. 249-264. New Orleans, LA, Jan. 1995.
 - [19] Seltzer, M. I., Ganger, G. R., McKusick, M. K., Smith, K. A., Soules, C. A. N., Stein, C. A., "Journaling vs. Soft Updates: Asynchronous Meta-data Protection in File Systems," *USENIX Annual Technical Conference*, San Diego, CA, June 2000.
 - [20] Stein, C. A., "The Write-Ahead File System: Integrating Kernel and Application Logging," Harvard CS Technical Report TR-02-2000, Cambridge, MA, Apr. 2000.
 - [21] Ylonen, T. "SSH—Secure Login Connections Over the Internet," *6th USENIX Security Symposium*, pp. 37-42. San Jose, CA, Jul. 1996.

The Multi-Queue Replacement Algorithm for Second Level Buffer Caches

Yuanyuan Zhou and James F. Philbin

NEC Research Institute, 4 Independence Way, Princeton, NJ 08540

Kai Li

Computer Science Department, Princeton University, Princeton, NJ 08544

Abstract

This paper reports our research results that improve second level buffer cache performance. Several previous studies have shown that a good single level cache replacement algorithm such as LRU does not work well with second level buffer caches. Second level buffer caches have different access pattern from first level buffer caches because *Accesses* to second level buffer caches are actually *misses* from first level buffer caches.

The paper presents our study of second level buffer cache access patterns using four large traces from various servers. We also introduce a new second level buffer cache replacement algorithm called Multi-Queue (MQ). Our trace-driven simulation results show that MQ performs better than all seven tested alternatives. Our implementation on a real storage system validates these results.

1 Introduction

Servers such as file servers, storage servers, and web servers play a critical role in today's distributed, multiple-tier computing environments. In addition to providing clients with key services and maintaining data consistency and integrity, a server usually improves performance by using a large buffer to cache data. For example, both EMC Symmetric Storage Systems and IBM Enterprise Storage Server deploy large software-managed caches to speed up I/O accesses [7, 8].

Figure 1 shows two typical scenarios of networked clients and servers. A client can be either an end user program such as a file client (Figure 1a), or a front-tier server such as a database server (Figure 1b). A server buffer cache thus serves as a

second level buffer cache in a multi-level caching hierarchy. In order to distinguish server buffer caches from traditional single level buffer caches, we call a server buffer cache a *second level buffer cache*. In contrast, we call a client cache or a front-tier server cache as a *first level buffer cache*.

Second level buffer caches have different access pattern from single level buffer caches because *accesses* to a second level buffer cache are *misses* from a first level buffer cache. First level buffer caches typically employ an LRU replacement algorithm, so that recently accessed blocks will be kept in the cache. As a result, accesses to a second buffer cache exhibit poorer temporal locality than those to a first level buffer cache; a replacement algorithm such as LRU, which works well for single level buffer caches, may not perform well for second level buffer caches.

Muntz and Honeyman [28] investigated multi-level caching in a distributed file system, showing that server caches have poor hit ratios. They concluded that the poor hit ratios are due to poor data sharing among clients. This study did not characterize the behavior of accesses to server buffer caches, but raised the question whether the algorithms that work well for client or single level buffer caches can effectively reduce misses for server caches. Willick, Eager and Bunt have demonstrated that the Frequency Based Replacement (FBR) algorithm performs better for file server caches than locality based replacement algorithms such as LRU, which works well for client caches [43]. However, several key related questions still remain open.

- Do other server workloads have access patterns similar to file servers?
- How do recently proposed client cache replacement algorithms such as LRU-k [15], Least Frequently Recently Used (LFRU) [14] and Two Queues (2Q) [23] perform for server caches?

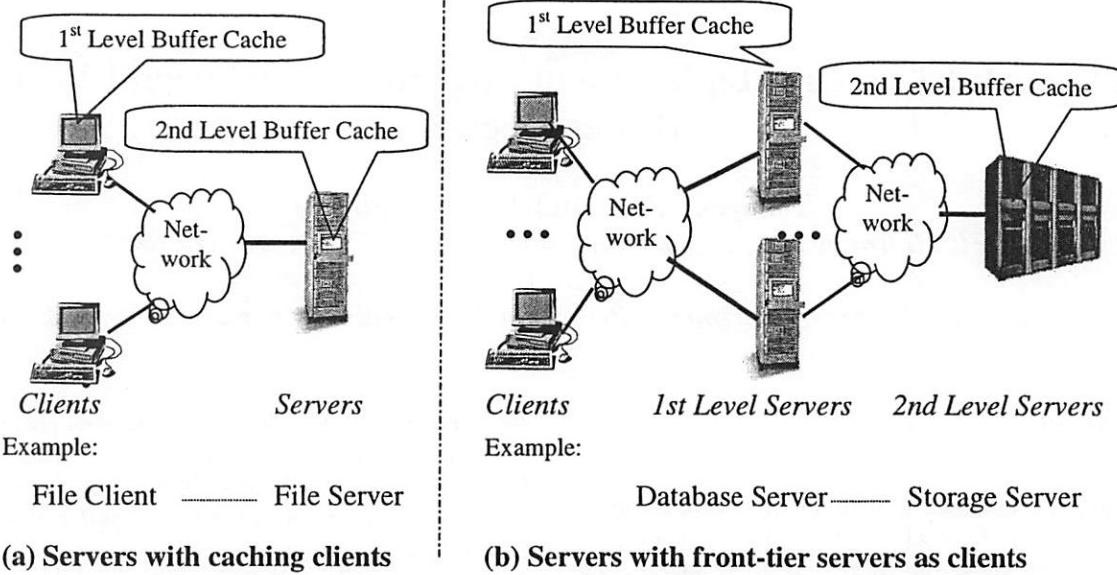


Figure 1: Second level buffer caches.

- What properties should a good server buffer cache replacement algorithm have?
- Is there an algorithm that performs better than FBR for server buffer caches?

This paper reports the results of our study of these questions. We first studied second level buffer cache access patterns using four large traces from file servers, disk subsystems and database back-end storage servers. Our analysis shows that a good second level buffer cache replacement algorithm should have three properties: minimal life time, frequency-based priority and temporal frequency. Finally, we introduce a new algorithm called Multi-Queue. Our trace-driven simulation results show that the new algorithm performs better than LRU, MRU, LFU, FBR, LRU-2, LFRU and 2Q, and that it is robust for different workloads and cache sizes. Our result also reveals that the 2Q algorithm, which does not perform as well as others for single level buffer caches, has higher hit ratios than all tested alternatives except Multi-Queue for second level buffer caches.

To further validate our results, we have implemented the Multi-Queue and LRU algorithms on a storage server system with the Oracle 8i Enterprise Server as the client. Our results using the TPC-C benchmark on a 100 GBytes database demonstrate that the Multi-Queue algorithm improves the transaction rate by 8-11% over LRU. To achieve the same improvement with LRU requires doubling the cache size of the storage server.

2 Methodology

The goal of our study is to improve the second level buffer cache performance. In this section, we briefly describe the existing algorithms tested in our evaluation and present the four traces used in our study.

2.1 Algorithms

We evaluate seven existing on-line replacement algorithms that were originally designed for client/single level buffer caches.

The **Least Recently Used (LRU)** algorithm has been widely employed for buffer cache management [9, 6]. It replaces the block in the cache which has not been used for the longest period of time. It is based on the observation that blocks which have been referenced in the recent past will likely be referenced again in the near future. However, for second level buffer caches, this observation is no longer present, or exists to much lesser extent. That is the reason why LRU does not perform well for file server caches in Willick, Eager and Bunt's study [43]. It is interesting to see whether this is also true for other workloads such as database back-end storage servers. The time complexity of this algorithm is $O(1)$.

The **Most Recently Used (MRU)** algorithm is also called Fetch-and-Discard replacement algorithm [9, 6]. This algorithm is used to deal with

the case such as sequential scanning access pattern, where most of recently accessed pages are not reused in the near future. Blocks that are recently accessed in a second level buffer cache will likely stay in a first level buffer cache for a period of time, so they won't be reused in the second level buffer cache in the near future. Does this mean MRU is likely to perform well for second level buffer caches? Willick, Eager and Bunt did not evaluate this algorithm in their study [43]. The time complexity of this algorithm is $O(1)$.

The **Least Frequently Used (LFU)** algorithm is another classic replacement algorithm. It replaces the block that is least frequently used. The motivation for this algorithm is that some blocks are accessed more frequently than others so that the reference counts can be used as an estimate of the probability of a block being referenced. The "aged" LFU usually performs better than the original LFU because the former gives different weight to the recent references and very old references. In [43], "Aged" LFU always performs better than LRU for the file server workload, except when the client cache is small compared to the second level buffer cache. Our results show this is true for two traces, but for the other four, LFU performs worse than LRU because they have some temporal locality. The time complexity of this algorithm is $O(\log(n))$.

The **Frequency Based Replacement(FBR)** algorithm was originally proposed by Robinson and Devarakonda [31] within a context of a stand-alone system. It is a hybrid replacement policy combining both LRU and LFU algorithms in order to capture the benefits of both algorithms. It maintains the LRU ordering of all blocks in the cache, but the replacement decision is primarily based upon the frequency count. The time complexity of this algorithm ranges from $O(1)$ to $O(\log_2 n)$ depending on the section sizes. The algorithm also requires parameter tuning to adjust the section sizes. So far, no on-line adaptive scheme has been proposed. In Willick, Eager and Bunt's file server cache study (1992) [43], FBR is the best algorithm among all tested ones including LRU, LFU, MIN, and RAND.

The **Least k th-to-last Reference (LRU- k)** algorithm was first proposed by O' Neil, et.al for database systems [15] in 1993. It bases its replacement decision on the time of the K^{th} -to-last reference of the block, i.e., the reference density observed during the past K references. When K is large, it can discriminate well between frequently and infrequently referenced blocks. When K is small, it can

remove cold blocks quickly since such blocks would have a wider span between the current time and the K th-to-last reference time. The time complexity of this algorithm is $O(\log(n))$.

The **Least Frequently Recently Used (LFRU)** algorithm was proposed by Lee, et.al in 1999 to cover a spectrum of replacement algorithms that include LRU at one end and LFU at the other [14]. It endeavors to replace blocks that are the least frequently used and not recently used. It associates a value, called Combined Recency and Frequency (CRF), with each block. The algorithm replaces the block with the minimum CRF value. Each reference to a block contributes to its CRF value. A reference's contribution is determined by a weighting function $F(x)$ where x is the time span from the reference to the current time. By changing the parameters of the weighting function, LFRU can implement either LRU or LFU. The time complexity of this algorithm is between $O(1)$ and $O(\log(n))$. It also requires parameter tuning and no dynamic scheme has been proposed.

The **Two Queue (2Q)** algorithm was first proposed for database systems by Johnson and Shasha in 1994 [23]. The motivation is to remove cold blocks quickly. It achieves this by using one FIFO queue A_{1in} and two LRU lists, A_{1out} and A_m . When first accessed, a block is placed in A_{1in} ; when a block is evicted from A_{1in} , its identifier is inserted into A_{1out} . An access to a block in A_{1out} promotes this block to A_m . The time complexity of the 2Q algorithm is $O(1)$. The authors have proposed a scheme to select the input parameters: A_{1in} and A_{1out} sizes. Lee and et. al. showed that 2Q does not perform as well as others for single level buffer caches [14]. However, our results show that 2Q in most cases performs better than tested alternatives except the new algorithm for second level buffer caches.

2.2 Traces

We have collected four server buffer cache traces from file servers, disk subsystems and database back-end storage servers. These traces are chosen to represent different types of workloads. All traces contained only misses from one or multiple client buffer caches that use LRU or its variations as their replacement algorithms. The block sizes for these traces are 8 Kbytes.

Table 1 shows the characteristics of the four traces.

	First Level Cache Size (MBytes)	# Reads (millions)	# Writes (millions)	# Clients or # First level Servers
Oracle Miss Trace-128M	128	7.3	4.3	single
Oracle Miss Trace-16M	16	3.8	2.0	single
HP Disk Trace	30	0.2	0.3	multiple
Auspex Server Trace	8 per client	1.8	0.8	multiple

Table 1: Characteristics of the four traces used in the study.

The first level buffer cache size clearly affects server buffer cache performance. We set the first level buffer cache sizes for the two Oracle traces to represent typical configurations in real systems. However, we could not change the first level buffer cache sizes of the other two traces because they were obtained from other sources.

Oracle Miss Trace-128M is collected from a storage system connecting to an Oracle 8i database client running the standard TPC-C benchmark [42, 27] for about two hours. The Oracle buffer cache replacement algorithm is similar to LRU [5]. The TPC-C database contains 256 warehouses and occupies around 100 GBytes of storage excluding log disks. The trace captures all I/O accesses from the Oracle process to the storage system. That is, the trace includes only reads that are missed on the Oracle buffer cache and writes that are flushed back to the storage system periodically or at commit time. The trace ignores all accesses to log disks. In order to better represent the workload on a real database system, we used 128 MBytes for the Oracle buffer cache.

Oracle Miss Trace-16M is collected with the same setup as the previous trace except the database buffer cache (first level buffer cache) size is set to 16 MBytes. For both traces, we fixed the execution time to be 2 hours instead of fixing the total number of transactions. Oracle Miss Trace-128M has performed many more transactions than the second trace. That is why both traces have similar amount of misses.

HP Disk Trace was collected at Hewlett-Packard Laboratories in 1992 [33, 32]. It captured all low-level disk I/O performed by the system. We used the trace gathered on Cello, which is a timesharing system used by a group of researchers at Hewlett-Packard Laboratories to do simulations, compilation, editing and mail. We have also tried other HP disk trace files, and the results are similar.

Auspex Server Trace was an NFS file system activity trace on an Auspex file server in 1993 at

UC Berkeley [16]. The system included 237 clients spread over four Ethernets, each of which connected directly to the central server. The trace covers seven days. We preprocessed the trace to include only block and directory read and write accesses.

Similarly to [16], we first split the trace into small trace files according to the client host ID. We then ran these traces through a multi-node cache simulator and collected the interleaved misses from different client caches as our server buffer cache trace. The multi-node client cache simulator uses 8 MBytes for each client cache and runs the LRU replacement algorithm.

3 Server Access Pattern

We have studied the access pattern of these four traces with the goal of understanding the behavior of server buffer cache accesses by examining their temporal locality and the distribution of access frequency.

3.1 Temporal Locality

The first part of our study explored the temporal locality of server buffer cache accesses. Past studies have shown that client buffer cache accesses exhibit a high degree of spatial and temporal locality. An accessed block exhibits temporal locality if it is likely to be accessed again in the near future. An accessed block exhibits spatial locality, if blocks near it are likely to be accessed in the near future [11, 38]. The LRU replacement algorithm, typically used in client buffer caches, takes advantage of temporal locality. Thus, blocks with a high degree of temporal locality are likely to remain in a client buffer cache, but accesses to a server buffer cache are misses from a client buffer cache. Do server buffer cache accesses exhibit temporal locality similar to those of a client buffer cache?

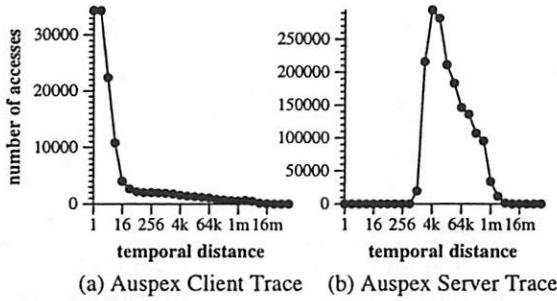


Figure 2: Comparison of temporal locality of client and server buffer cache accesses using temporal distance histograms. (Note: figures are in different scales)

We used *temporal distance* histograms to observe the temporal locality of the server buffer cache traces. A *reference sequence* (or *reference string*) is a numbered sequence of temporally ordered accesses to a server buffer cache. The *temporal distance* is the distance between two accesses to the same block in the reference sequence. It is similar to the inter-reference gap in [30]. For example, in the reference sequence *ABCDBAX*, the temporal distance from A_1 to A_6 is 5 and the temporal distance from B_2 to B_5 is 3. Formally speaking, if we denote the sequence number of the current access and previous access to a block b as *current*(b) and *prev*(b) respectively, then the temporal distance of the current access to block b is *current*(b) – *prev*(b). A temporal distance histogram shows how many *correlated accesses* (accesses to the same block) for various temporal distances reside in a given reference sequence.

Figure 2 compares the client and server buffer cache's temporal locality using temporal distance histograms with the Auspex traces. The client buffer cache trace is collected at an Auspex client, while the server buffer cache trace is captured at the Auspex File Server. Each Auspex client uses an 8 megabyte buffer cache. The data in the figure shows the histograms by grouping temporal distances by powers of two. The block size is 8 Kbytes. Results are similar with other block sizes. Distances that are not a power of two are rounded up to the nearest power of two. Significantly, for the client buffer cache 74% of the correlated references have a temporal distance less than or equal to 16. This indicates a high degree of temporal locality. Even more significantly, however, 99% of the correlated accesses to the server buffer cache have a temporal distance of 512 or greater, exhibiting much weaker temporal locality.

Figure 3 shows the temporal distance histograms of four server buffer cache traces. These traces exhibit two common patterns. First, all histogram curves are hill-shaped. Second, peak temporal distance values, while different, are all relatively large and occur at distances greater than their client cache sizes (see Table 1). This access behavior at server buffer caches is expectable. If a client buffer cache of size k uses an locality-based replacement policy, after a reference to a block, it takes at least k references to evict this block from the client buffer cache. Thus, subsequent accesses to the server buffer cache should be separated by at least k non-correlated references in the server buffer cache reference sequence.

A good replacement algorithm for server buffer caches should retain blocks that reside in the “hill” portion of the histogram for a longer period of time. In this paper, “time” means logical time, measured by the number of references. For example, initially, time is 0, after accesses *ABC*, time is 3. We call the beginning and end of this “hill” region *minimal distance* (or *minDist*) and *maximal distance* (or *maxDist*) respectively. We picked *minDist* and *maxDist* for each trace by examining the histogram figure for simplicity. Since the temporal distance values in the “hill” are relatively large, a good replacement algorithm should keep most blocks in this region for at least the *minDist* time. We call this property *minimal lifetime property*. It is clear that when the number of blocks in a server buffer cache is less than the *minDist* of a given workload, the LRU policy tends to perform poorly, because most blocks do not stay in the server buffer cache long enough for subsequent correlated accesses.

3.2 Access Frequency

Next, we examined the behavior of server buffer cache accesses in terms of frequency. While it is clear that server buffer cache accesses represent misses from client buffer caches, the distribution of access frequencies among blocks remains uncertain. If the distribution is even, then most replacement algorithms will perform similarly to or worse than LRU. If the distribution is uneven, then a good replacement algorithm will keep frequently accessed blocks in a server buffer cache. Past studies [11, 38] have shown that blocks are typically referenced unevenly: a few blocks are hot (frequently accessed), some blocks are warm, and most blocks are cold (infrequently accessed). Is this also true for server buffer caches?

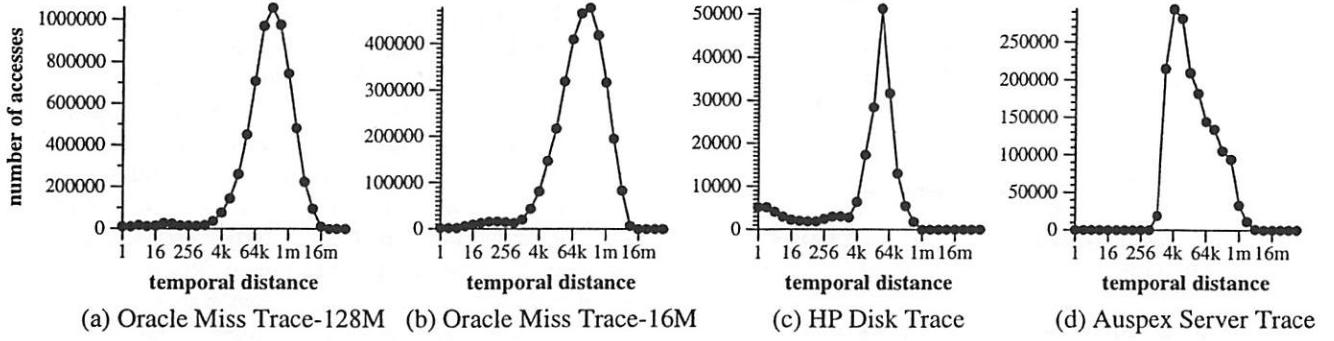


Figure 3: Temporal distance histograms of server buffer cache accesses for different traces. (Note: figures are in different scales)

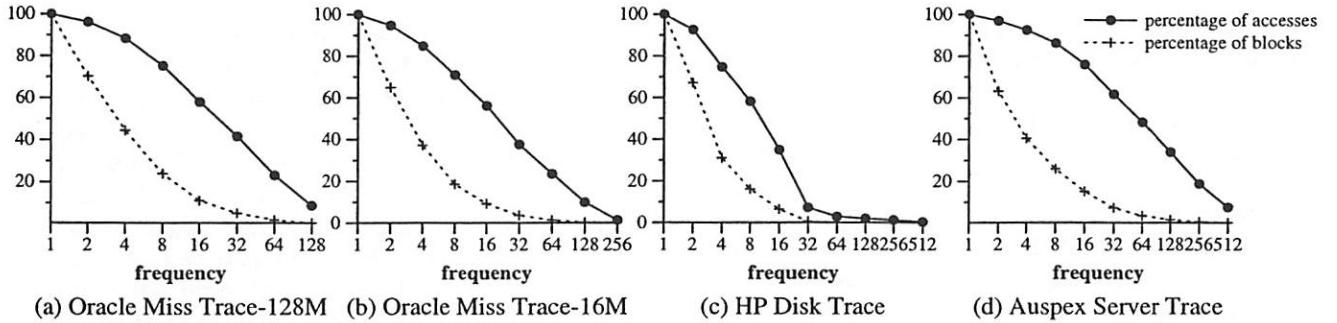


Figure 4: Access and block distribution among different frequencies. A point (f, p_1) on the block percentage curve indicates that $p_1\%$ of total number of blocks are accessed at least f times, while a point (f, p_2) on the access percentage curve represents that $p_2\%$ of total number of accesses are to blocks accessed at least f times.

Our hypothesis is that both hot and cold blocks will be referenced less frequently in server buffer caches, because hot blocks will stay in client buffer caches most of the time and cold blocks will be accessed infrequently by definition. If this hypothesis is true, the access frequency distributions at server buffer caches should be uneven, though probably not as uneven as those at client buffer caches. A good server buffer cache replacement algorithm should be able to identify warm blocks and keep them in server buffer caches for a longer period of time than others.

In order to understand the frequency distributions of reference sequences seen at server buffer caches, we examined the relationship between access distribution and block distribution for different frequencies. Similar to most cache studies, frequency here means the number of accesses. Figure 4 shows, for a given frequency f , the percentage of total number of blocks accessed at least f times. It also shows the percentage of total accesses to those types of blocks. Notice that the number of blocks accessed at least i times includes blocks accessed at least j times ($j > i$). This explains why all the curves always decrease gradually. The access percentage curves decrease similarly for the same reason.

For all four traces, the access percentage curves decrease more slowly than the block percentage curves, indicating that a large percentage of accesses are to a small percentage of blocks. For example, in the Oracle Miss Trace-128M, around 60% accesses are made to less than 10% blocks, each of which are accessed at least 16 times. This shows that the access frequency distribution among blocks at server buffer caches is uneven. In other words, a subset of blocks are accessed more frequently than others even though the average temporal distance between two correlated accesses in this subset is very large (Figure 3). Thus, if the replacement algorithm can selectively keep those blocks for a long period of time, it will significantly reduce the number of misses, especially when the server buffer cache size is small.

3.3 Properties

To summarize our study results, a good server buffer cache replacement algorithm should have the following three properties:

1. **Minimal lifetime:** warm blocks should stay in a server buffer cache for at least $minDist$

time for a given workload.

2. **Frequency-based priority:** blocks should be prioritized based on their access frequencies.
3. **Temporal frequency:** blocks that were accessed frequently in the past, but have not been accessed for a relatively long time should be replaced.

The first two properties are derived from our study of the traces. The third property is obvious. It addresses the common access pattern where a block is accessed very frequently for some time and then has no accesses for a relatively long time.

Algorithms developed in the past do not possess all three properties. Both LRU and MRU algorithms satisfy the temporal frequency property, but lack the other two. The basic LFU algorithm possesses only the second property. With frequency aging, it can satisfy the third. LRU-2 can satisfy the third property but it only partially satisfies the first and second. FBR and LFRU vary between LRU and LFU depending on the input parameters, but it is almost impossible to find parameters that satisfy all three properties at once. 2Q satisfies the third property, but it can only partially satisfy the second property. When the server buffer cache is small, 2Q lacks the first property, but, for large cache sizes, satisfies it.

4 Multi-Queue Algorithm

We have designed a new replacement algorithm, called Multi-Queue (MQ), that satisfies the three properties above. This algorithm maintains blocks with different access frequencies for different periods of time in the second level buffer cache.

The MQ algorithm uses multiple LRU queues: Q_0, \dots, Q_{m-1} , where m is a parameter. Blocks in Q_j have a longer lifetime in the cache than those in Q_i ($i < j$). MQ also uses a history buffer Q_{out} , similarly to the 2Q algorithm [23], to remember access frequencies of recently evicted blocks for some period of time. Q_{out} only keeps block identifiers and their access frequencies. It is a FIFO queue of limited size.

On a cache hit to block b , b is first removed from the current LRU queue and then put at the tail of queue Q_k according to b 's current access frequency. In

other words, k is a function of the access frequency, $QueueNum(f)$. For example, for a given frequency f , $QueueNum(f)$ can be defined as $\log_2 f$. So the 8th access to a block that is already in the second level buffer cache will promote this block from Q_2 to Q_3 according to this $QueueNum(f)$ function.

On a cache miss to block b , MQ evicts the head of the lowest non-empty queue from the second level buffer cache in order to make room for b , i.e. MQ starts with the head of queue Q_0 when choosing victims for replacement. If Q_0 is empty, then MQ evicts the head block of Q_1 , and so on. If block c is the victim, its identifier and current access frequency are inserted into the tail of the history buffer Q_{out} . If Q_{out} is full, the oldest identifier in Q_{out} will be deleted. If the requested block b is in Q_{out} , then it is loaded and its frequency f is set to be the remembered value plus 1, and then b 's entry is removed from Q_{out} . If b is not in Q_{out} , it is loaded into the cache and its frequency is set to 1. Finally, block b is inserted into an LRU queue according to the value of $QueueNum(f)$.

MQ demotes blocks from higher to lower level queues in order to eventually evict blocks that have been accessed frequently in the past, but have not been accessed for a long time. MQ does this by associating a value called *expireTime* with each block in the server buffer cache. "Time" here refers to logical time, measured by number of accesses. When a block stays in a queue for longer than a permitted period of time without any access, it is demoted to the next lower queue. This is easy to implement with LRU queues. When a block enters a queue, the block's *expireTime* is set to be *currentTime* + *lifeTime*, where *lifeTime*, a tunable parameter, is the time that each block can be kept in a queue without any access. At each access, the *expireTime* of each queue's head block is checked against the *currentTime*. If the former is less than the latter, it is moved to the tail of the next lower level queue and the block's *expireTime* is reset. Figure 5 gives a pseudo-code outline for the MQ algorithm.

When m equals 1, the MQ algorithm is the LRU algorithm. When m equals 2, the MQ algorithm and the 2Q algorithm [23] both use two queues and a history buffer. However, MQ uses two LRU queues, while 2Q uses one FIFO and one LRU queue. MQ demotes blocks from Q_1 to Q_0 when their life time in Q_1 expires, while 2Q does not make this kind of adjustment. When a block in Q_1 (or A_m) is evicted in the 2Q algorithm, it is not put into the history

```

/* Procedure to be invoked upon a reference
to block b */
if b is in cache{
    i = b.queue;
    remove b from queue Q[i];
} else{
    victim = EvictBlock();
    if b is in Qout {
        remove b from Qout;
    } else{
        b.reference = 0;
    }
    load b's data into victim's place;
}
b.reference++;
b.queue = QueueNum(b.reference);
insert b to the tail of queue Q[k];
b.expireTime = currentTime + lifeTime;
Adjust();

EvictBlock(){
    i = the first non-empty queue number;
    victim = head of Q[i];
    remove victim from Q[i];
    if Qout is full
        remove the head from Qout;
    add victim's ID to the tail of Qout;
    return victim;
}

Adjust(){
    currentTime++;
    for(k=1; k<m; k++){
        c = head of Q[k];
        if(c.expireTime < currentTime){
            move c to the tail of Q[k-1];
            c.expireTime = currentTime + lifeTime;
        }
    }
}

```

Figure 5: MQ algorithm

buffer whereas it is with MQ.

Like the 2Q algorithm, MQ has a time complexity of $O(1)$ because all queues are implemented using LRU lists and m is usually very small (less than 10). At each access, at most $m - 1$ head blocks are examined for possible demotion. MQ is faster in execution and also much simpler to implement than algorithms like FBR, LFRU or LRU-K, which have a time complexity close to $O(\log_2 n)$ (where n is the number of entries in the cache) and usually require a heap data structure for implementation.

MQ satisfies the three properties that a good second level buffer cache replacement algorithm should have. It satisfies the minimal lifetime property because warm blocks are kept in high level LRU queues for at least $expireTime$ time, which is usually greater than the $minDist$ value of a given workload. It satisfies the frequency-based priority property because blocks that are accessed more frequently are put into higher level LRU queues and are, therefore, less likely to be evicted. It also sat-

isfies the temporal frequency property because MQ demotes blocks from higher to lower level queues when its lifetime in its current queue expires. A block that has not been accessed for a long time will be gradually demoted to queue Q_0 and eventually evicted from the second level buffer cache.

5 Simulation and Results

This section reports our trace-driven simulation results of nine replacement algorithms including LRU, MRU, LFU, LRU-2, FBR, LFRU, 2Q, OPT (an optimal off-line algorithm), and MQ. Our goal is to answer three questions:

- How does MQ compare with other algorithms?
How does recently proposed single level cache replacement algorithms such as LRU-2, LFRU and 2Q perform for second level buffer caches?
- How can we use the second level buffer cache access behaviors to explain the performance?
- How do one use the simulation information to tune the performance of the MQ algorithm?

The following addresses each question in turn.

5.1 Simulation Experiments

We have implemented the nine replacement algorithms in our buffer cache simulator. The block size for all simulations is 8 KBytes. With experiments, we found out that using $\log(f)$ function as our $QueueNum(f)$ function works very well for all traces. Our experiments also show that eight LRU queues are enough to separate the warmer blocks from the others. The history buffer Q_{out} size is set to be four times of the number of blocks in the cache. Each entry of the history buffer occupies fewer than 32 bytes so that the memory requirement for the history buffer is quite small, less than 0.5% of the buffer cache size. The $lifeTime$ parameter is adjusted adaptively at running time. The main idea for dynamic $lifeTime$ adjusting is to efficiently collect statistic information on the temporal distance distributions from access history. Due to page limits, we will not discuss it in this paper, but details can be found in [44].

The history buffer size for 2Q is one half of the number of blocks in the cache as suggested by Johnson and Shasha in [23]. For fairness, we have extended the LFRU, LRU-2, FBR and LFU algorithms to use a history buffer to keep track of *CRF* values, second-to-last reference time and access frequencies for recently evicted blocks respectively (see section 2), using a history buffer of size equal to that in MQ. We have tuned the FBR and LFRU algorithms with several different parameters as suggested by the authors and report the best performance. The off-line optimal algorithm (OPT) was first proposed by Belady [2, 17] and is widely used to derive the lower bound of cache miss ratio for a given reference string. This algorithm replaces the block with the longest future reference distance. Since it relies on the precise knowledge of future references, it cannot be used on-line.

ws Belady's OPT algorithm and WORST algorithm [2, 17]

As with all cache studies, interesting effects can only be observed if the size of the working set exceeds the cache capacity. The three traces provided by other sources (HP Disk Trace, Auspex Server Trace and Web Server Trace) have relatively small working sets. To anticipate the current trends that working set sizes increase with user demands and new technologies, we chose to use smaller buffer cache sizes for these three traces. In most of experiments, we set the second level buffer cache size to be larger than the first level buffer cache size. However, this property does not always hold in real systems. For example, most of storage systems such as the IBM Enterprise Storage Server have less than 1 Giga Bytes of storage cache (second level buffer cache), while the frontier server, database or file servers, typically have more than 2 Gigabytes of buffer cache (first level buffer cache). Because of this reason, we have also explored a few cases where the second level buffer cache is equal to or smaller than the first level buffer cache.

5.2 Results

Table 2 shows that the MQ algorithm performs better than other on-line algorithms. Its performance is robust for different workloads and cache sizes. MQ is substantially better than LRU. With the Oracle Miss Trace-128M, LRU's hit ratio is 30.9% for a 512 Mbytes server cache, whereas MQ's is 47.5%, a 53% improvement. For the same cache size, MQ has a

10% higher hit ratio than FBR. The main reason for MQ's good performance is that this algorithm can selectively keep warm blocks in caches for a long period of time till subsequent correlated accesses.

LRU does not perform well for the four server cache access traces, though it works quite well for client buffer caches. This is because LRU does not keep blocks in the cache long enough. The LFU algorithm performs worse than LRU. The long temporal distance (*minDist*) at server buffer caches makes frequency values inaccurate. Of the eight on-line algorithms, the MRU algorithm has the worst performance. Although this algorithm can keep old blocks for a long time in server buffer caches, it does not consider frequencies. As a result, some blocks kept in server buffer caches for a long time are not accessed frequently.

FBR, LFRU and LRU-2 perform better than LRU but always worse than MQ. The gap between these three algorithms and MQ is quite large in several cases. Although FBR and LFRU can overcome some of the LRU drawbacks by taking access frequency into account, it is difficult to choose the right combination of frequency and recency by tuning the parameters for these two algorithms. LRU-2 does not work well because it favors blocks with small temporal distances.

2Q performs better than other on-line algorithms except MQ. With a separate queue ($A_{1,n}$) for blocks that have only been accessed once, 2Q can keep frequently accessed blocks in the A_m queue for a long period of time. However, when the server buffer cache size is small, 2Q performs worse than MQ. For example, with Oracle Miss Trace-128M, 2Q has a 4% lower hit ratio than MQ for a 512 MBytes cache. With Oracle Miss Trace-16M, the gap between MQ and 2Q is 6.7% for a 64 MBytes cache. This is because the lifetime of a block in the 2Q server buffer cache is not long enough to keep the block resident for the next access.

5.3 Performance Analysis

To understand the performance results in more detail, we use temporal distance as a measure to analyze the algorithms. Since the traces in our study have similar access patterns, this section reports the analysis using the Oracle Miss Trace-128M trace as a representative.

The performance of a replacement algorithm at

Cache Size	OPT	MQ	2Q	FBR	LFRU	LRU2	LRU	LFU	MRU
64MB	21.6	14.0	12.0	8.4	10.1	8.1	6.1	5.9	2.6
128MB	30.3	21.7	20.0	14.6	16.3	14.1	10.1	10.8	3.7
256MB	41.8	33.0	30.0	24.2	24.3	23.5	17.6	18.7	5.8
512MB	56.1	47.5	43.5	37.8	39.5	38.2	30.9	31.1	9.9
1GB	70.7	62.1	62.1	55.8	58.8	57.2	53.0	47.6	17.9
2GB	82.0	76.3	76.8	75.2	76.8	75.8	74.5	65.1	33.7

(a) Oracle Miss Trace-128M

Cache Size	OPT	MQ	2Q	FBR	LFRU	LRU2	LRU	LFU	MRU
16MB	16.5	10.0	7.5	4.4	5.1	4.2	4.1	3.2	1.9
32MB	22.7	15.2	12.4	9.0	10.0	7.2	6.3	6.0	2.3
64MB	30.8	22.9	16.2	15.5	19.0	12.6	11.4	11.0	3.1
128MB	40.8	32.3	32.5	25.2	26.8	21.5	19.9	19.1	4.7
256MB	52.4	44.1	43.8	38.4	36.0	34.0	32.2	30.3	7.9
512MB	63.9	57.4	57.8	53.7	50.2	49.5	47.7	44.5	14.3
1GB	72.6	69.2	69.1	68.1	67.0	66.0	64.8	61.1	26.7
2GB	83.8	80.1	80.0	79.7	80.1	79.5	79.2	76.1	50.1

(b) Oracle Miss Trace-16M

Cache Size	OPT	MQ	2Q	FBR	LFRU	LRU2	LRU	LFU	MRU
16MB	36.5	22.0	20.6	20.5	20.2	12.9	14.5	20.2	12.6
32MB	49.9	36.4	36.3	30.0	29.6	24.8	22.1	29.6	16.9
64MB	65.8	54.2	53.8	47.4	44.5	47.6	41.4	43.6	22.7
128MB	77.2	68.9	69.2	65.1	65.0	64.0	62.5	57.3	32.9
256MB	81.2	78.5	78.3	78.5	78.0	76.8	77.8	71.5	51.0

(c) HP Disk Trace

Cache Size	OPT	MQ	2Q	FBR	LFRU	LRU2	LRU	LFU	MRU
8MB	32.4	21.7	9.4	8.4	8.4	13.1	2.0	6.9	0.8
16MB	45.2	33.0	31.3	19.2	18.7	26.5	16.7	13.8	1.8
32MB	57.7	47.1	46.9	38.7	38.3	40.3	36.1	20.7	3.5
64MB	68.7	59.3	59.5	55.5	55.0	53.4	53.3	25.7	7.3
128MB	77.9	70.5	70.4	68.3	67.3	64.9	66.9	35.4	13.9
256MB	86.4	81.3	80.8	80.9	78.8	76.3	78.0	60.6	26.3

(d) Auspex Server Trace

Table 2: Hit ratios in percentage

Algorithms	distance < 64k		distance \geq 64k	
	#hits	#misses	#hits	#misses
MQ	1553k	293k	1919k	2646k
2Q	1846k	0	1330k	3235k
FBR	1611k	234k	1146k	3418k
LFRU	1412k	434k	1470k	3094k
LRU2	1606k	239k	1179k	3385k
LRU	1846k	0	407k	4157k
LFU	1077k	769k	1196k	3368k
MRU	285k	1560k	434k	4131k

Table 3: Oracle Miss Trace-128M hits and misses distribution with a 512 MBytes cache (Note: first-time accesses to any blocks are not counted in either category).

server caches primarily depends on how well they can satisfy the life time property. As we have observed from Section 3, accesses to server caches tend to have long temporal distances. If the majority of accesses have temporal distances greater than D , a replacement algorithm that cannot keep blocks longer than D time is unlikely to perform well.

Our method to analyze the performance is to classify all accesses into two categories according to their temporal distances: $< C$ and $\geq C$ where C is the number of entries in the server buffer cache. Table 3 shows the number of hits and misses in the two categories for a 512 MBytes server buffer cache.

LRU has no miss in the left category because any access in this category is less than C references away

from its previous access to the same block. The block being accessed should still remain in the cache since the buffer cache can hold C blocks. However, LRU has a large number of misses in the right category because any block that has not been accessed for more than C time can be evicted from the cache and therefore lead to a miss for the next access to this block. Since the right category dominates the total number of accesses (Figure 3(a)), LRU does not perform well.

The 2Q, FBR, LFRU and LRU2 algorithms reduce the number of misses in the right category by 15-25% because these algorithms can keep warm blocks in the cache longer than C time. However, in order to achieve this, the FBR, LFRU and LRU2 algorithms have to sacrifice some blocks, which are kept in the cache for a short period of time. As a result, these three algorithms have some misses in the left category. But the number of such misses is much smaller than the number of misses avoided in the right category. Overall, the three algorithms have fewer misses than LRU. Because the 2Q algorithm has no misses in the left category, it outperforms the FBR, LFRU and LRU2 algorithms.

MQ significantly reduces the number of misses in the right category. As shown on Table 3, MQ has 2,646k misses in the right category, 36% fewer than LRU. Similarly to the FBR algorithm, MQ also has some misses in the left category. However, the number of such misses is so small that it contributes to only 10% of the total number of misses. Overall,

the MQ algorithm performs better than other online algorithms.

6 Implementation and Results

We have implemented the MQ and LRU algorithms in a storage server system. The goals of our implementation are:

- to validate the simulation results;
- to study the end performance improvement on a real system.

This section describes the storage system architecture, the MQ and LRU implementation, the experiment setups, and the experimental results of the TPC-C benchmark with the Oracle 8i Enterprise Server.

6.1 Architecture

We have implemented a storage server system using a PC cluster. The storage system manages multiple virtual volumes (virtual disks). A virtual volume can be implemented using a single or multiple physical disk partitions. Similarly to other clustered storage system [35], our storage system runs on a cluster of four PCs. Each PC is 400 MHz Pentium II with 512 KBytes second level cache and 1 GB of main memory. All PCs are connected together using Giganet [22]. Clients communicate with storage server nodes using the Virtual Interface (VI) communication model [29]. The peak communication bandwidth is about 100 MBytes/sec and the one-way latency for a short message is about 10 microseconds [22]. Data transfer from Oracle's buffer to the storage buffer uses direct DMA without memory copying. Each PC runs Windows NT 4.0 operating system. The interrupt time for incoming messages is 20 microseconds. Each PC box holds seven 17 GBytes IBM SCSI disks, one of which is used for storing the operating system. The bandwidth of data transfers between disk and host memory is about 15 Mbytes/sec and the access latency for random read/writes is about 9 milliseconds. Each PC in our storage system has a large buffer cache to speed up I/O accesses.

We have implemented both MQ and LRU as the cache replacement algorithms. The parameters of

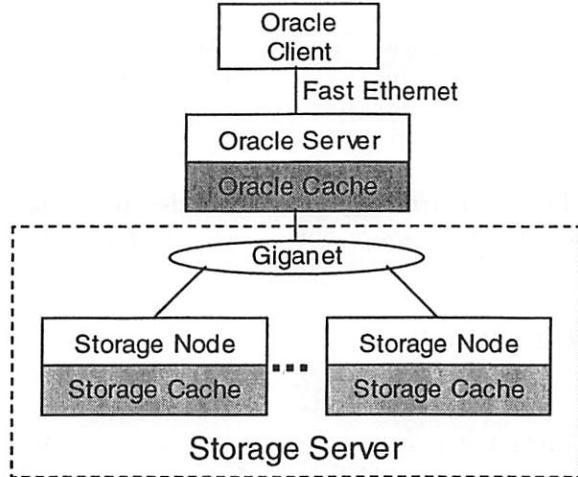


Figure 6: The architecture of a storage server.

the MQ algorithm are the same as described in the previous section. It uses eight queues. The history buffer size is four times the number of cache blocks. The *lifeTime* parameter is set dynamically after the warmup phase and adjusted periodically using the statistic information [44].

We measure the performance using the TPC-C benchmark [27] running on the Oracle 8i Enterprise Server [5]. Figure 6 gives the architecture of our experiments. The hardware and software setups are similar to those used for collecting the Oracle Miss Trace-128M. The Oracle 8i Server runs on a separate PC, serving as a client to the storage system. It accesses raw partitions directly. All raw I/O requests from the Oracle server are forwarded to the storage system through Giganet. The Oracle buffer cache is configured to be 128 MBytes. Other parameters of the Oracle Server are well tuned to achieve the best TPC-C performance. Each test runs the TPC-C script on an Oracle client machine for 2 hours. The Oracle client also runs on a separate PC which connects to the Oracle server through Fast Ethernet. The TPC-C script is provided by the Oracle Corporation. It simulates 48 clients, each of which generates transactions to the Oracle server. The TPC-C benchmark emulates the typical transaction processing of warehouse inventories. Our database contains 256 warehouses and occupies 100 GBytes disk space excluding logging disks. Logging disk data is not cached in the storage system. The storage system employs a write-through cache policy.

Storage cache size	MQ	LRU
128MB	19.85	8.85
256MB	31.42	17.66
512MB	44.34	31.69

Table 4: Percentage hit ratios of the storage buffer cache. The Oracle buffer cache (first level buffer cache) size is always 128 Mbytes.

6.2 Results

Table 4 shows the hit ratios of the storage buffer cache with the MQ and LRU replacement algorithms. The difference between the implementation and simulation results is less than 10%, which validates our simulation study. The small difference is mainly caused by two reasons. The first is that the timing is different in the real system due to concurrency. The second is the interaction between cache hit ratios and request rates. When the cache hit ratio increases, the average access time decreases. As a result, more I/O requests are forwarded to the storage system.

As shown on Table 4, MQ achieves much higher hit ratios than LRU. For a 512 MBytes storage buffer cache, MQ has a 12.65% higher hit ratio than LRU. Since miss penalty dominates the average access time, we use the relative miss ratio difference to estimate the upper bound of MQ's improvement on the end performance. For a 512 MBytes buffer cache, the relative miss ratio difference between MQ and LRU is 18.5% ($12.65/(100 - 31.69)$). Therefore, the upper bound for end performance improvement with MQ over LRU is 18.5%.

In fact, in order for LRU to achieve the same hit ratio as MQ, its cache size needs to be doubled. The hit ratio of MQ with a 128 MBytes cache is slightly greater than that of LRU with a 256 MBytes cache. The hit ratio of MQ with a 256 MBytes cache is about the same as LRU with a 512 MBytes cache.

Figure 7 shows the end performance of the MQ and LRU algorithms. For all three buffer cache sizes, MQ improves the TPC-C end performance over LRU by 8-11%. Due to certain license problems, we are not allowed to report the absolute performance in terms of transaction rate. Therefore, all performance numbers are normalized to the transaction rate with a 128 MBytes buffer cache using the LRU replacement algorithm. Because of the high hit ratios, the MQ algorithm improves the transac-

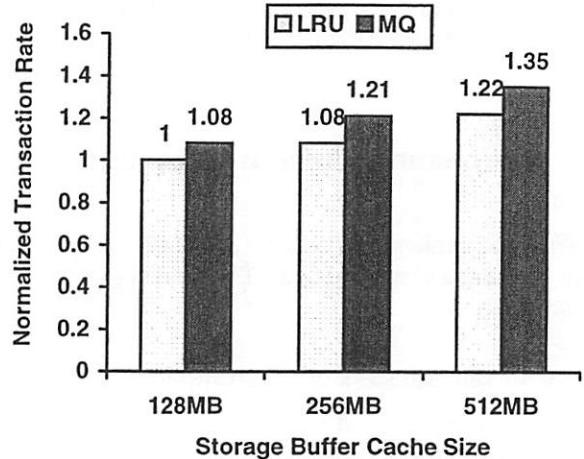


Figure 7: Normalized TPC-C transaction rate with different storage buffer cache sizes (All numbers are normalized to the transaction rate achieved by LRU with an 129 MBytes storage buffer cache).

tion rates over LRU by 8%, 12%, and 10% for 128 MBytes, 256 MBytes and 512 MBytes cache sizes respectively.

Similar to the cache hit ratio improvement, using the MQ algorithm is equivalent to using LRU with a double sized cache. With a 128 MBytes buffer cache, MQ increases the transaction rate by 8%, which is exactly same as the improvement achieved by LRU with a 256 MBytes buffer cache. MQ with a 256 MBytes cache achieves a similar transaction rate to LRU with a 512 MBytes cache.

7 Related Work

A large body of literature has examined cache replacement algorithms. Examples of buffer cache replacement algorithms include the LRU [9, 6], GCLOCK [36, 19], First in First Out (FIFO), MRU, LFU, Random, FBR [31], LRU- k [15], 2Q [23], and LFRU [14]. In the spectrum of off-line algorithms, Belady's OPT algorithm and WORST algorithm [2, 17] are widely used to derive a lower and upper bound on the cache miss rate. Other closely related works include Muntz and Honeyman's file server caching study [28] and Eager and Bunt's disk cache study [43]. Most of these works have been described in our Introduction and Methodology sections.

Cache replacement policies have been intensively studied in various contexts in the past, including

processor caches [38], paged virtual memory systems [36, 3, 40, 4, 12, 4, 34, 13, 10], and disk caches [37]. Although several studies [1, 20, 24] focus on two level processor cache design issues, their conclusions do not apply to software based L2 buffer cache designs because the former has more restrictions. Some analytical models of the storage hierarchies have been given in [21, 25].

Many past studies have used metrics such as LRU stack distance [17], marginal distribution of stack distances [18] or distance string models [39] to analyze the temporal locality of programs. However, the proposed LRU stack distance models were designed specifically for stack replacement algorithms like LRU. Moreover, distance string models do not capture the long-range relationships among references. O’Neil and et. al. recently proposed the inter-reference gap (IRG) model [30] to characterize temporal localities in program behavior. The IRG value for an address in a trace represents the time interval between successive references to the same address. But this model looks at each address separately and does not look at the overall distribution of the IRG values. Therefore, it cannot well capture global access behavior.

Our study uses the distribution of temporal distances to measure temporal locality. The idea of using multiple queues with feedback has appeared in process scheduling [26, 41]. With this method, the priority of a process increases on an I/O event and decreases when its time slice expires without an I/O event.

8 Conclusions

Our study of second level buffer cache access patterns has uncovered two important insights. First, accesses to server buffer caches have relatively long temporal distances, unlike those to client buffer caches, which are much shorter. Second, access frequencies are distributed unevenly; some blocks are accessed significantly more often than others.

These two insights helped us identify three key properties that a good server buffer cache replacement algorithm should possess: *minimal lifetime, frequency-based priority, and temporal frequency*. Known replacement algorithms such as LRU, MRU, LFU, FBR, LRU-2, LFRU, and 2Q do not satisfy all three properties; however, our new algorithm,

Multi-Queue (MQ), does.

Our simulation results show that the MQ algorithm performs better than other on-line algorithms and that it is robust for different workloads and cache sizes. In particular, MQ performs substantially better than the FBR algorithm which was the best algorithm in a previous study [43]. In addition, another interesting result of our study is that the 2Q algorithm, which does not perform as well as other algorithms for single level buffer caches [14], outperforms them for second level buffer caches, with the exception of MQ.

We have implemented the Multi-Queue and LRU algorithms on a storage server using an Oracle 8i Enterprise Server as the client. The results of the TPC-C benchmark on a 100 GBytes database show that the MQ algorithm has much better hit ratios than LRU and improves the TPC-C transaction rate by 8-12% over LRU. For LRU to achieve a similar level of performance, the cache size needs to be doubled.

Our study has two limitations. First, we implemented only MQ and LRU replacement algorithms on our storage system. It would be interesting to compare these with other algorithms. Second, this paper assumes that the only information an L2 buffer cache algorithm has is the misses from the L1 buffer cache. It is conceivable that the L1 buffer cache might pass hints to the L2 cache in addition to the misses themselves. We have not explored this possibility.

References

- [1] J.-L. Baer and W.-H. Wang. On the Inclusion Properties for Multi-Level Cache Hierarchies. In *15th ISCA*.
- [2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2), 1966.
- [3] R. W. Carr and J. L. Hennessy. WSClock - A Simple and Effective Algorithm for virtual Memory Management. In *SOSP*, 1981.
- [4] H. T. Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems. In *VLDB*, 1985.
- [5] Oracle Co. *Oracle 8i Concepts*.
- [6] E. G. Coffman and P. J. Denning. *Operating Systems Theory*. 1973.
- [7] EMC Corporation. Symmetrix 3000 and 5000 Enterprise Storage Systems Product Description Guide. 1999.

- [8] IBM Corporation. White Paper: ESS-The Performance Leader. 1999.
- [9] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5), May 1968.
- [10] P. J. Denning. Virtual Memory. *ACM Computing Surveys*, 28(1), March 1996.
- [11] P. J. Denning and S. C. Schwartz. Properties of the working-set model. *Communications of the ACM*, 15(3), March 1972.
- [12] W. Effelsberg and T. Haerder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, 9(4), December 1984.
- [13] C. Lee et.al. HiPEC: High Performance External Virtual Memory Caching. In *1st OSDI*, 1994.
- [14] D. Lee et.al. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *SIGMETRICS-99*, pages 134–143, 1999.
- [15] E.J. O’Neil et.al. The LRU-K Page Replacement Algorithm For Database Disk Buffering. In *SIGMOD-93*, 1993.
- [16] M. D. Dahlin et.al. A Quantitative Analysis Scalability for Network File Systems. In *SIGMETRICS-94*, 1994.
- [17] R. L. Mattson et.al. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2), 1970.
- [18] V. Almeida et.al. Characterizing reference locality in the WWW. In *PDIS-96*, 1996.
- [19] V. F. Nicola et.al. Analysis of the Generalized Clock Buffer Replacement Scheme for Database Transaction Processing. In *SIGMETRICS-92*, 1992.
- [20] W.-H. Wang et.al. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. In *16th ISCA*, 1989.
- [21] J. Gecsei and J. A. Lukes. A Model for the Evaluation of Storage Hierarchies. *IBM Systems Journal*, 13(2):163–178, 1974.
- [22] Giganet Inc. Giganet.
- [23] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB-94*, 1994.
- [24] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th ISCA*, 1990.
- [25] C. Lam and S. E. Madnick. Properties of Storage Hierarchy Systems with Multiple Page Sizes and Redundant Data. *ACM Transactions on Database Systems*, 4(3):345–367, September 1979.
- [26] B. W. Lampson. A Scheduling Philosophy for Multiprocessing Systems. *Communications of the ACM*, 11(5), 1968.
- [27] S. T. Leutenegger and D. Dias. A modeling study of the TPC-C benchmark. *SIGMOD Record*, 22(2), June 1993.
- [28] D. Muntz and P. Honeyman. Multi-level Caching in Distributed File Systems -or- your cache ain’t nuthin’ but trash. In *Proceedings of the Usenix Winter Technical Conference*, 1992.
- [29] VI Architecture Organization. Virtual Interface Architecture Specification version 1.0. 1997.
- [30] V. Phalke and B. Gopinath. An Inter-Reference Gap Model for Temporal Locality in Program Behavior. In *SIGMETRICS-95*, 1995.
- [31] J. Robinson and M. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *SIGMETRICS-90*, 1990.
- [32] C. Ruemmler and J. Wilkes. A Trace-Driven Analysis of Disk Working Set Sizes. Technical Report HPL-OSR-93-23, Hewlett-Packard Laboratories, Palo Alto, CA, USA, April 5 1993.
- [33] C. Ruemmler and J. Wilkes. UNIX Disk Access Patterns. In *USENIX-93*, 1993.
- [34] G. M. Sacco and M. Schkolnick. Buffer Management in Relational Database Systems. *ACM Transactions on Database Systems*, 11(4), December 1986.
- [35] R. A. Shillner and E. W. Felten. Simplifying distributed file systems using a shared logical disk. Technical Report TR-524-96, Princeton University CS Department, 1996.
- [36] A. J. Smith. Sequentiality and Prefetching in Database Systems. *ACM Transactions on Database Systems*, 3(3), September 1978.
- [37] A. J. Smith. Disk cache - miss ratio analysis and design considerations. *TOCS*, 3, 1985.
- [38] Alan J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.
- [39] J. R. Spinn. Distance string models for program behavior. *Computer*, 9(11), November 1976.
- [40] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7), July 1981.
- [41] A. S. Tanenbaum. *Modern Operating Systems*. 1992.
- [42] Transaction Processing Performance Council. *TPC Benchmark C*. May 1991.
- [43] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk Cache Replacement Policies for Network File-servers. In *ICDCS*, May 1993.
- [44] Yuanyuan Zhou. Memory Management for Networked Servers (Thesis). Technical report, Princeton University, Computer Science Department, November 2000.

Design and Implementation of a Predictive File Prefetching Algorithm

Thomas M. Kroeger[†]

*Nokia Cluster IP Solutions
Santa Cruz, California*

Darrell D. E. Long[‡]

*Jack Baskin School of Engineering
University of California, Santa Cruz*

Abstract

We have previously shown that the patterns in which files are accessed offer information that can accurately predict upcoming file accesses. Most modern caches ignore these patterns, thereby failing to use information that enables significant reductions in I/O latency. While prefetching heuristics that expect sequential accesses are often effective methods to reduce I/O latency, they cannot be applied across files, because the abstraction of a file has no intrinsic concept of a successor. This limits the ability of modern file systems to prefetch. Here we present our implementation of a predictive prefetching system, that makes use of file access patterns to reduce I/O latency.

Previously we developed a technique called *Partitioned Context Modeling* (PCM) [13] that efficiently models file accesses to reliably predict upcoming requests. We present our experiences in implementing predictive prefetching based on file access patterns. From the lessons learned we developed of a new technique *Extended Partitioned Context Modeling* (EPCM), which has even better performance.

We have modified the Linux kernel to prefetch file data based on *Partitioned Context Modeling* and *Extended Partitioned Context Modeling*. With this implementation we examine how a prefetching policy, that uses such models to predict upcoming accesses, can result in large reductions in I/O latencies. We tested our implementation with four different application-based benchmarks and saw I/O latency reduced by 31% to 90% and elapsed time reduced by 11% to 16%.

1 Introduction

The typical latency for accessing data on disk is in the range of tens of milliseconds. When compared to the 2 nanosecond clock step of a typical 500 megahertz processor, this is very slow (5,000,000 times slower). The result is that I/O cache misses will force fast CPUs to sit idle while waiting for I/O to complete. This difference between processor and disk speeds is referred to as the I/O gap [22]. Prefetching methods based on sequential heuristics are only able to partially address the I/O gap, leaving a need for more intelligent methods of prefetching file data.

Caching recently accessed data is helpful, but without prefetching its benefits can be limited. By loading data in anticipation of upcoming needs, prefetching can turn a 20 millisecond disk access into a 100 microsecond page cache hit. This is why most I/O systems prefetch extensively based on a sequential heuristic. For example, disk controllers frequently do *read-ahead* (prefetching of the next disk block), and file systems often prefetch the next sequential page within a file. In both of these cases, prefetching is a heuristic guess that accesses will be sequential and can be done because there is sequential structure to the data abstraction. However, once a file system reaches the end of a file it typically has no notion of “next file” and is unable to continue prefetching.

Despite this lack of sequential structure there are still strong relationships that exist between files and cause file accesses to be correlated. Several studies [12, 13, 6, 23, 16, 3, 18, 25] have shown that predictable reference patterns are quite common, and offer enough information for significant performance improvements. Previously, we used traces of file system activity to demonstrate the extent of the relationships between files [13]. These traces covered all system calls over a one month period on four separate machines. From these traces we observed that a simple *last successor* prediction model (which predicts that an access to file A will be followed by the same file that followed the last access to A) correctly predicted 72% of file access events. We

[†]tmk@cips.nokia.com. Supported in part by the Usenix Association and the National Science Foundation under Grant CCR-9704347.

[‡]darrell@cse.ucsc.edu. Supported in part by the National Science Foundation under Grant CCR-9704347.

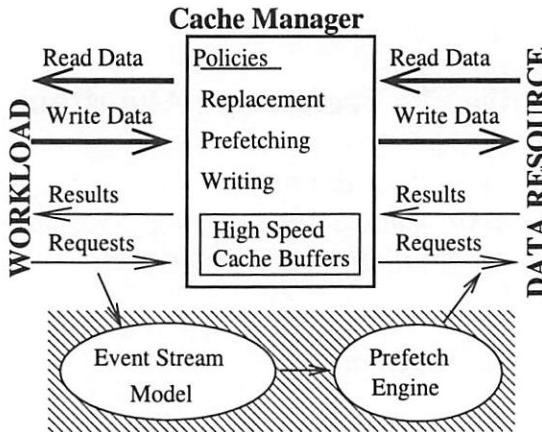


Figure 1: Cache system with predictive prefetching.

also presented a more accurate technique called *Partitioned Context Modeling (PCM)* that efficiently handles the large number of distinct files and adapts to changing reference patterns.

To demonstrate the effectiveness of using file access relationships to improve I/O performance we added predictive prefetching to the Linux kernel. We enhanced the normal Linux file system cache by adding two components, a *model* that tracks the file reference patterns and a *prefetch engine* that uses the model's predictions to select and prefetch files that are likely to be requested. Figure 1 illustrates how these components integrate into an I/O cache (the shaded area indicates the new components).

To evaluate our implementations we used four application-based benchmarks, the compile phase of the Andrew benchmark [7], the linking of the Linux kernel, a Glimpse text search indexing [20] of the `/usr/doc` directory, and a compiling, patching and re-compiling of the SSH source code versions 1.2.18 through 1.2.31. For both last successor and Partitioned Context Model (PCM) based prefetching, we observed that predicting only the next event limited the effectiveness of predictive prefetching. To address this limitation, we modified PCM to create a variation called *Extended Partitioned Context Modeling (EPCM)*, which predicts sequences of upcoming accesses, instead of just the next access. Our tests showed that EPCM based predictive prefetching reduced I/O latencies, from 31% to as much as 90%, and that total elapsed time was reduced by 11% to 16%. We concluded that a predictive prefetching system has the potential to significantly reduce I/O latency and is effective in improving overall performance.

2 Modeling Background

The issues of file access pattern modeling have been explored previously [12, 13, 14, 15]. We present a brief background on the three modeling techniques used in our implementation: last-successor, Partitioned Context Modeling and Extended Partitioned Context Modeling.

The last-successor model was used as a simple baseline for comparison. This model predicts that an access to file A will be followed by an access to the same file that followed the last access to A. This model requires only one node per unique file so we can say that its state space is $O(n)$, where n is the number of unique files. We saw that for a wide variety of file system traces this last successor model was able to correctly predict the next access an average of 72% of the time [13].

2.1 Context Modeling

Partitioned Context Modeling originated from Finite Multi-Order Context Modeling (FMOCM) and the text compression algorithm PPM [2]. A context model is one that uses preceding events to model the next event. For example, in the string “object” the character “t” is said to occur within the context “objec”. The length of a context is termed its *order*. In the example string, “jec” would be considered a third order context for “t”. Techniques that predict using multiple contexts of varying orders (*e.g.* “ec”, “jec”, “bjec”) are termed *Multi-Order Context Models* [2]. To prevent the model from quickly growing beyond available resources, most implementations of a multi-order context model limit the highest order modeled to some finite number m , hence the term *Finite Multi-Order Context Model*. In these examples we have used letters of the alphabet to illustrate how this modeling works in text compression. For modeling file access patterns, each of these letters is replaced with a unique file.

A context model uses a *trie* [11], a data structure based on a tree, to efficiently store sequences of symbols. Each node in this trie contains a symbol (*e.g.* a letter from the alphabet, or the name of a specific file). By listing the symbols contained on the path from the root to any individual node, each node represents an observed pattern. The children of every node represent all the symbols that have been seen to follow the pattern represented by the parent. To model access probabilities we add to each node a count of the number of times that pattern has been seen. By comparing the counts of the sequence just seen

with the counts of those nodes that previously followed this pattern we can generate predictions of what file will be accessed next.

Figure 2 extends an example from Bell *et al.* [2] to illustrate how this trie would develop when given the sequence of events *CACBCAABCA*. In this diagram the circled node *A* represents the pattern *CA*, which has occurred three times. This pattern has been followed once by another access to the file *A* and once by an access to the file *C*. The third time is the last event to be seen and we haven't yet seen what will follow. We can use this information to predict both *A* and *C* each with a likelihood of 0.5. The state space for this model is proportional to the number of nodes in this tree, which is bounded by $O(n^m)$, where m is the highest order tracked and n is number of unique files. On a normal file system were the number of files can range between 10 thousand and 100 million such space requirements are unreasonable. In response, we developed the Partitioned Context Model (PCM).

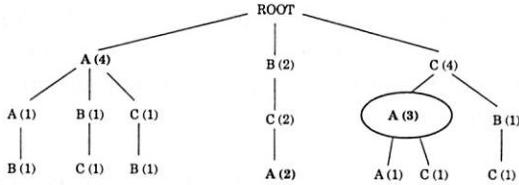


Figure 2: Example trie for the sequence *CACBCAABCA*.

2.2 Partitioned Context Modeling (PCM)

To address the space requirements of *FMOCM*, we developed the Partitioned Context Model. This model divides the trie into partitions, where each partition consists of a first order node and all of its descendants. The number of nodes in each partition is limited to a static number that is a parameter of the model. The effect of these changes is to reduce the model space requirements from $O(n^m)$ to $O(n)$. Figure 3 shows the trie from Figure 2 with these static partitions.

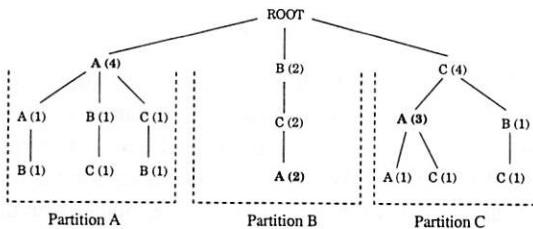


Figure 3: Example partitioned trie for the access sequence *CACBCAABCA*.

When a new node is needed in a partition that is full, all

node counts in the partition are divided by two (integer division), any nodes with a count of zero are cleared to make space for new nodes. If no space becomes available, the access is ignored. Another benefit of restricting space in this manner is that when new access patterns occur, existing node counts decay exponentially, causing the model to adapt faster to new access patterns. While PCM solves the space problem, our experiments showed that it did not predict far enough into the future to give time for the prefetch to complete, so we developed EPCM.

2.3 Extended PCM (EPCM)

Our initial test with Last Successor and PCM prefetching on the Andrew benchmark test showed that the predictions were occurring too close to the time that the data was actually needed. Specifically the *prefetch lead* time—the time between the start of a data prefetch and the actual workload request for that data—was much smaller than the time needed to read the data from disk. In fact, last successor based prefetching running the Andrew benchmark made correct predictions an average of 1.23 milliseconds before the data was needed. On the other hand, file data reads took on the order of 10 milliseconds. This lead time of 1.23 milliseconds severely limited the potential gains from the last successor based prefetching.

To address the need for more advance notice of what to prefetch, we modified PCM to create a technique called Extended Partition Context Modeling. This technique extends the model's maximum order to approximately 75% to 85% of the partition size and restricts how the partition grows by only allowing one new node for each instance of a specific pattern, similar to how Lempel-Ziv [26] encoding builds contexts. In this technique, the patterns modeled grow in length by one node each time they occur. When we predict from an EPCM model, just as with PCM, we use a given context's children to predict the next event. In addition, if the predicted node has a child that has a high likelihood of access, we can also predict that file. This process can continue until the descendant's likelihood of access goes below the prefetch threshold. For our context models we set a prefetch threshold as the minimum likelihood that a file must have in order to be prefetched. As long as this threshold is greater than 0.5 then each level can predict at most one file, and EPCM will predict the sequence of accesses that is about to occur.

Figure 4 shows an example extended partition. In this

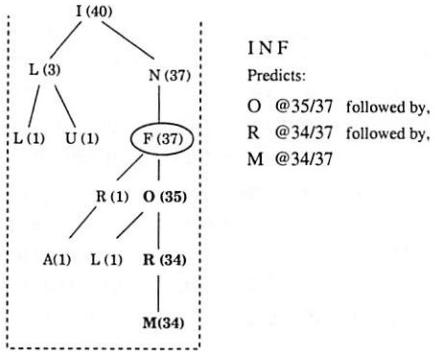


Figure 4: An example EPCM partition.

example, the circled node is a third order context that represents the sequence **INF**. From this partition, as with PCM, we see that the sequence **INF** has occurred 37 times, and it has been followed by an access to the file **O** 35 times. However, we can also see that 34 of those times **O** was followed by **R** and then **M**. So, this model will predict that the sequence **ORM** will be followed by the sequence **ORM** each node will have their counts incremented and the event seen following **M** will be added as a child of **M**.

3 Implementation

In order to gain a more complete insight into how our prefetching methods would interact with a typical I/O system, we implemented predictive prefetching in the Linux kernel. Our implementation consisted of adding two components to the VFS layer of the Linux kernel, a model and a prefetching engine. Our models tracked file access patterns and produced a set of predictions for upcoming accesses. Our prefetch engine selected predicted files and prefetched their data into the page cache. The implementation itself consisted of less than 2000 lines of C code.

3.1 The Linux Kernel's VFS Layer

The *Virtual File System* (VFS) layer [1] provides a uniform interface for the kernel to deal with various I/O requests and specifies a standard interface that each file system must support. Through this layer, one kernel can mount several different types of file systems (*e.g.* *EXT2FS*, *ISO9660FS*, *NFS*, ...) into the same tree structure. We worked with version 2.2.12 of the Linux kernel

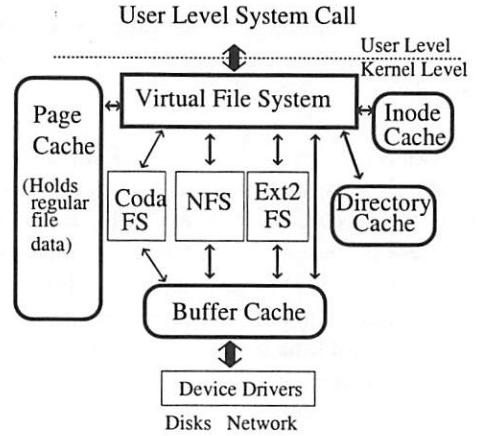


Figure 5: The Linux kernel's I/O caches.

and confined our changes to the VFS layer. By doing all of our changes in the VFS layer we kept our predictive prefetching totally independent of the underlying file system.

Arguably, the most important service the VFS layer provides is a uniform I/O data cache. Linux maintains four caches of I/O data: *page cache*, *i-node cache*, *buffer cache* and *directory cache*. Figure 5 shows these caches and how they interact with the kernel, each other and user level programs. The *page cache* combines virtual memory and file data. The *i-node cache* keeps recently accessed file i-nodes. The *buffer cache* interfaces with block devices, and caches recently used meta-data disk blocks. The Linux kernel reads file data through the buffer cache, but keeps the data in the page cache for reuse on future reads. The *directory cache* (d-cache) keeps in memory a tree that represents a portion of the file system's directory structure. This tree maps a file's path name to an i-node structure and speeds up file path name look up. The basic element of the d-cache is a structure called the *d-entry*.

We implemented our methods of modeling file access patterns by adding one field to the *d-entry* structure. The various models would attach their modeling data structure to this pointer. For the last successor model this consisted of just a device and inode number. For the partitioned models this was a pointer to the partition that began with the file that the *d-entry* identified. After each file access the model would update its predictions. The prefetch engine was then called and would use these predictions to prefetch file data.

4 Evaluating Predictive Prefetching

Here we present the results from our benchmark tests on predictive prefetching and how they affected the design of our implementation. We ran our tests on a Pentium based machine with a SCSI I/O subsystem and 256 megabytes of RAM. To evaluate our implementation we selected four application based benchmarks that provide a variety of workloads. In our test we saw predictive prefetching reduce the time spent waiting for I/O by 31% to 90%. While read latencies saw reductions from 33% to 92%, the reductions in elapsed time, ranged from 11% to 16%.

Our test machine had a Pentium Pro 200 CPU, with 256 megabytes of RAM, an Adaptec AHA-2940 Ultra Wide SCSI controller and a Seagate Barracuda (ST34371W) disk. All kernels were compiled without symmetric multi-processor (SMP) support. This machine had Gnu **ld** version 2.9.1.0.19, **gcc** version 2.7.2.3 and Glimpse version 4.1.

For these tests, we focused primarily on two measures—the read latency and total I/O latency. We determined read latency from instrumentation of the read system call. Since this did not include I/O latencies from page faults, open events, and exec calls, we also considered the total I/O latency. We bound total I/O latency by taking the difference between the elapsed time and the amount of time the benchmark was computing (time in the running state or system time plus user time). This gives us the amount of time that the benchmark spent in a state other than running, which served as an upper bound on the amount of time spent waiting on I/O. Since our test machines had only the bare minimum of daemon processes, this measure is a close approximation of the total I/O latency of that benchmark.

Each test consisted of 3 warm up runs that eliminated initial transient noise and allowed the models time to learn. Then 20 runs of the test benchmark provided enough samples for us to generate meaningful confidence intervals assuming a normal distribution and statistically significant measurements. Unless otherwise stated, the I/O caches were cleared between each run of the benchmark.

4.1 Measuring Predictive Prefetching

Accurately measuring the effectiveness of predictive prefetching presented a significant problem in itself.

Most file system benchmarks such as PostMark [9] use a randomly generated workload. Since our work is based on the observation that file access patterns are not random these benchmarks offer little potential for measuring predictive prefetching. In fact, many researchers [6, 12, 13, 23, 16, 3, 18, 25] have shown that this random workload incorrectly represents file system activity.

Previously [13], we used traces of file system activity over a one month period from four different machines to show that PCM based predictions can predict the next access with an accuracy of 0.82. Across the four traces the accuracy measures ranged from 0.78–0.88. These four traces were chosen to represent the most diverse set of I/O characteristics from the 33 different machines traced. Even with the widest range of I/O characteristics possible the one characteristic that was uniform across all traces was predictability. Unfortunately, most existing benchmarks lack any such predictability.

Replaying our traces on a live system was another method we considered for testing predictive prefetching. While these traces did contain a record of all system calls, page fault data was not recorded. Unfortunately one common source for I/O requests is page faults that result from memory mapped executables and data files. As a result, an application based benchmark which consisted of executing specific programs (and the associated page faults) would more accurately represent a realistic file system workload.

For these reasons we choose to use application based benchmarks to provide a basic but realistic measure of how well predictive prefetching would do under some well defined conditions. While these benchmarks don't represent a real world workload, they do provide a workload that is more realistic than that of random file access benchmarks or replayed traces. To provide enough data samples to obtain confidence intervals of our measures we ran each benchmark 20 times. While such repetition lacks the additional variety that would occur in many real world workloads, this workload is similar to those seen by a nightly build process or the traversal of a set of data files (*e.g.* indexing of man pages).

Finally, we should note that predictive prefetching suffers from the same compulsory miss problems that an LRU cache does. Specifically, if our system hasn't previously seen an access pattern then there is no way it can recognize that pattern, predict a file's access and prefetch the file's data. This means that any meaningful benchmark must see the given pattern at least once before it can recognize it. As a result we must train on an

access pattern to a set of files before we can meaningfully test predictive prefetching over that pattern. Our SSH benchmark addresses this concern by changing the source code base across several versions without any re-training. Thus measuring the performance of our predictive prefetching system over a changing code base.

4.2 Andrew Benchmark

Phase five of the Andrew benchmark [7] features a basic build of a C program. Although this benchmark is quite dated, to our knowledge it is the only existing file system benchmark that has been widely used and accurately portrays the predictive relationship between files. For these reasons our first benchmark was the build from phase five of the Andrew benchmark [7].

Initially predictive prefetching kernels were able to reduce the total I/O latency for this benchmark by 26%. From these tests we observed that to achieve greater reductions in I/O latency, our models would need to predict further ahead than merely the next event. So we modified PCM to create Extended Partitioned Context Modeling. Prefetching based on EPCM, was able reduce the total elapsed time by 12%, and remove almost all (90%) of the I/O latency from this benchmark.

4.2.1 Characterizing the Workload

The Andrew benchmark consists of five phases, however, the only phase that contained testing relevant to predictive prefetching is phase five, the compile phase. So when we refer to the Andrew benchmark we are referring to phase five of this benchmark. This test consists of compiling 17 C files and linking the created object files into two libraries and one executable program. The total source code consists of 431 kilobytes in 11,215 lines of code.

Tables 1 and 2 show summaries of time and event count statistics for the Andrew benchmark on the test machine under the unmodified Linux 2.2.12 kernel. The rows marked *Cold* represent tests where the I/O caches were cleared out prior to each run of the benchmark, while the rows marked *Hot* represent tests where the I/O caches were not cleared out. Note that the hot cache test required no disk accesses because all of the data for the Andrew benchmark was kept within the I/O caches on the test machine.

Table 1: Workload time summary for phase five of the Andrew benchmark. Elapsed and compute times are in seconds; read times are in microseconds. Numbers in *italics* represent 90% confidence intervals.

Test	Elapsed 90%	Compute 90%	Read 90%
Cold	9.15 <i>0.05</i>	7.94 <i>0.01</i>	646 <i>31.06</i>
Hot	7.95 <i>0.02</i>	7.93 <i>0.00</i>	139 <i>0.31</i>

Table 2: Read event count summary for the Andrew benchmark. Counts are the number of events that fell in that category averaged across the last 20 runs of the each test.

Test	Calls	Hits	Partial	Misses
Cold	919	334	537	47
Hot	919	382	537	0

Table 1 shows latency statistics. The column marked *Elapsed* represents the mean elapsed time for that test. The column marked *Compute* represents the amount of time the benchmark process was computing: the sum of the user time and system time for that test. This time represents a lower bound on how fast we can make our benchmark run. The column marked *Read* shows the average duration of read system calls. A 90% confidence interval follows each of these measures.

Table 2 shows read event count statistics. We divided read calls into three categories, hits, partial hits and misses. Hits required no disk access: data was already available in the page or buffer cache. Partial hits represent cases where the necessary data was already in the process of being read, but wasn't yet available. Misses represent events where the data request required new disk activity.

The Andrew benchmark workload is I/O intensive. However, many of the events are satisfied from the I/O caches. On our test machine this workload consisted of 919 read events, of which 47 required disk access with a cold cache, a miss ratio of 0.05. From the cold cache test we can see that it spent 7.94 seconds in the running state and it had a total elapsed time of 9.15 seconds. So we can bound its total I/O latency to at most the difference of these two numbers, which is 1.21 seconds for this case.

4.2.2 Initial Results

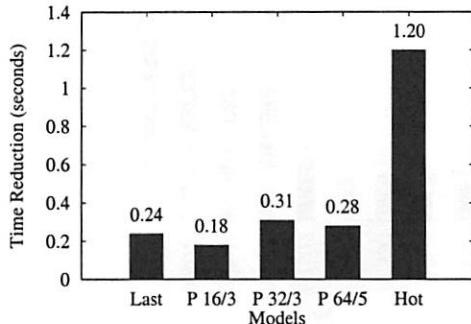
We ran the Andrew benchmark under kernels modified to prefetch based on PCM and last successor modeling. Figure 6 shows the elapsed time and read latency reductions for several tests. From these tests we saw reductions of up to 26% in total I/O latency and 15% in read latency. The simple last successor based prefetching did better than some settings of the more complex PCM based prefetching. PCM based prefetching improved as the partition size increases from 16 to 32, but the increase to 64 offered no further improvements.

However, the compute times for our benchmark tests increased 0.05 seconds, apparently due to modeling and prefetching overhead. Compute time for the last successor test increased by as much as, and in some cases more than, those for PCM based prefetching, even though last successor is a much simpler model. This indicates that the prefetching engine is most likely the dominant factor in the increased computational overhead. Latencies for both open and exec events also increased. Despite these increases, predictive prefetching reduced both the total I/O latency and read latency.

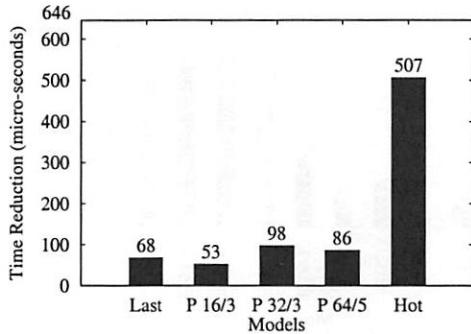
4.2.3 EPCM Results

To address the need for a greater prefetch-lead time we modified our PCM kernel to implement EPCM based prefetching. Figure 7(a) shows the results from EPCM base prefetching compared with those from the previous section. From this graph we see that EPCM based prefetching reduced our elapsed times by 1.11 seconds or 12%. While this is a modest gain in total elapsed time for the benchmark, it is a significant reduction when one recalls that the best reduction possible is 1.21 seconds of I/O latency. Thus with EPCM based prefetching we reduced the time this benchmark spent waiting on I/O by 90%.

Figure 7(b) shows the results for the read latency reduction from EPCM based prefetching. The latencies for read system calls with EPCM based prefetching are as low as 127 microseconds, a reduction in read latency of 80%. This latency is less than the 139 microsecond latencies for the hot cache test. The EPCM based prefetching does better than the hot cache test because of how Linux 2.2.12 does not write data directly from the page cache, and must transfer data to the buffer for writing. The first part of the Andrew benchmark creates object files. As these files are written, they are moved from the page cache to the buffer cache. During the linking



(a) Elapsed Time Reduction

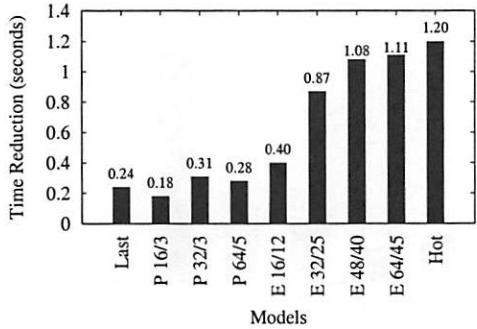


(b) Read Latency Reduction

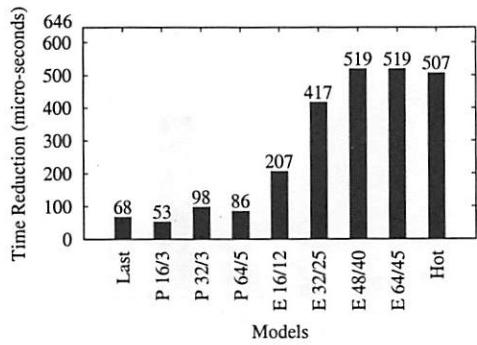
Figure 6: Reductions in elapsed times and read latencies for the Andrew benchmark with the last successor, PCM and hot cache tests. Bars marked with P represent PCM tests. Partition sizes (ps) and model order (mo) are labeled as ps/mo.

phase, we read all of this object file data. In the hot cache case, each read system call must copy the data from buffers in the buffer cache to a new page in the page cache. This buffer copy is time consuming. For files that are prefetched, this copy is done during the prefetch engine's execution and not during the read system call.

Figure 8 shows the distribution of read events for a typical hot cache test and a typical EPCM based prefetching test. The hot cache test has significantly more events that occur in the 129–256 microsecond bucket, while the EPCM test appears to account for that difference in 17–32 and 33–64 microsecond buckets. In other words, it appears many of the read system calls have become about 100 to 200 microseconds shorter as a result of the prefetching. In fact, during the selected hot cache run of the Andrew benchmark, we observed 1993 copies from the buffer cache to the page cache during read system calls. Since the predictive prefetching tests would do these buffer copies during their open and exec events the read system calls for those tests would not need to do



(a) Elapsed Time Reduction



(b) Read Latency Reduction

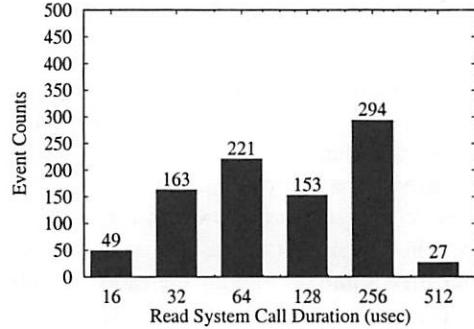
Figure 7: Reductions in elapsed times and read latencies for the Andrew benchmark with the last successor, PCM, EPCM and hot cache tests. Bars marked with P and E represent PCM and EPCM tests respectively. Partition sizes (ps) and model order (mo) are labeled as ps/mo.

these buffer copies. The result is that for this test on this kernel our predictive prefetching test has a lower read latency than that of the hot cache test where all the data is already in memory. This buffer copy problem has been fixed in version 2.4 of the Linux kernel.

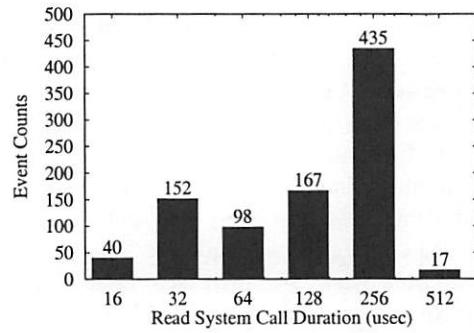
4.3 Linking the Linux Kernel

Our next benchmark of file system activity adapts a test used by Chang *et al.* [5] that focuses on the Gnu linker. A significantly larger workload than the Andrew benchmark, this workload consists of primarily non-sequential file accesses to temporary files. Our predictive prefetching was able to reduce the total I/O latency of this benchmark by as much as 34%, and again reduced the total runtime by 11%.

This test used the Linux kernel source and linked together all of the top level modules (*e.g.* **fs.o**, **mm.o**,



(a) EPCM ps 64 or 45 Test



(b) Hot Cache Test

Figure 8: Read system call latency distributions for selected runs of the Andrew benchmark (times in microseconds).

net.o, **kernel.o** ...) which were then linked into a final kernel image. It linked a total of 180 object files through 51 commands to create a kernel image of approximately twelve megabytes. Tables 3 and 4 show the summary statistics for our Gnu ld benchmark's workload. The cold cache test of our Gnu ld benchmark took approximately 36 seconds, with about 24 seconds of compute time for a 65% CPU utilization. We observed a miss ratio of 0.12. The latency for read events is significantly higher than those of the Andrew benchmark. The Gnu linker does not access individual files sequentially. This foils Linux's sequential read-ahead within each file and explains the high average read latencies, despite the low cache miss ratio. Additionally, the files being read in this benchmark are object files which are typically temporary in nature. As a result it is quite possible that the disk placement of these object files is not contiguous.

Figure 9 shows the results for our Gnu ld benchmark. These results are consistent with those seen from the Andrew benchmark. Although not as dramatic, we still saw significant reductions in total I/O latency and read latencies. Again, these reductions increase as model or-

Table 3: Workload time summary for the Gnu Id benchmark. Elapsed times are in seconds, read times are in microseconds. Numbers in italics represent 90% confidence intervals.

Test	Elap. 90%	Compute 90%	Read 90%
Cold	36.12 <i>0.13</i>	23.96 <i>0.03</i>	2866 <i>18.84</i>
Hot	23.98 <i>0.01</i>	23.95 <i>0.01</i>	596 <i>3.12</i>

Table 4: Read event count summary for the Gnu Id benchmark. Counts are the number of events that fell in that category averaged across the last 20 runs of the each test.

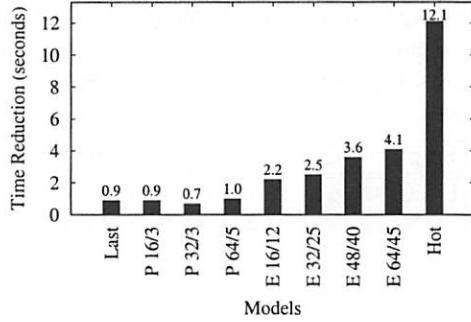
Test	calls	hits	partial	misses
Cold	6362	4794	767	799
Hot	6362	5694	668	0

der and partition size increase. PCM and last successor based prefetching do better than the normal Linux kernel with as much as a 8% reduction in the total I/O latency. The advanced predictions of EPCM seem to again offer a more substantial reduction of 34%. The reductions for read system calls are also not as astounding as those of the Andrew benchmark. Nevertheless, 33% reductions in read latencies are still a welcome improvement.

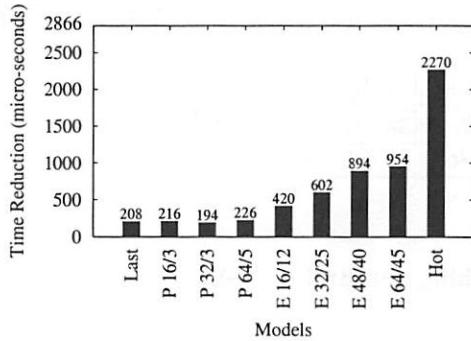
4.4 Glimpse Indexing

For our third benchmark we used a glimpse [20] index of **/usr/doc** to represent a traversal of all the files under a given directory. This workload is significantly larger than either of the two previously studied. For this benchmark we saw similar result to those from the Gnu Id benchmark. Specifically, the total benchmark runtime was reduced by 16%, the total I/O latency was reduced by 31% and read latencies were reduced by 92%.

The workload created by the **glimpseindex** program is a linear traversal of all the files in a large directory structure. We used version 4.1 of Glimpse and performed an index of **/usr/doc**. The order of files in their directory determines the order in which files are accessed. The large majority of files see only one access and are typically static files created when Linux was installed and have not been modified since. By comparison, access order in the Andrew benchmark's workload was dependent on the Makefile and the order in which header files were listed. Additionally, files such as header files and object files were accessed multiple times.



(a) Elapsed Time Reduction



(b) Read Latency Reduction

Figure 9: Reductions in elapsed times and read latencies for the Gnu Id benchmark with the last successor, PCM, EPCM and hot cache tests. Bars marked with P and E represent PCM and EPCM tests respectively. Partition sizes (ps) and model order (mo) are labeled as ps/mo.

Tables 5 and 6 show the workload characteristics for the glimpse benchmark on our test machine. This workload contains significantly more disk accesses, a total of 24,901 reads. A much higher fraction of these reads are cache misses, 11,812 misses for a miss ratio of 0.47. The hot cache test has cache misses, indicating that this test accesses more data than the I/O caches can hold.

Figure 10 shows the results for the glimpse benchmark. We saw the best results from the smallest EPCM test, reducing total runtime by 16%, read latencies reduced by as much as 92% and I/O latency by 31%. Our PCM test had a 22% reduction for this workload. The test of last successor based prefetching did the worst with an average total I/O latency reduction of 16%. Again we see the predictive prefetching has the potential for significant reductions in I/O latency and is effective at improving overall system performance.

Table 5: Workload time summary for the glimpse benchmark. Elapsed times are in seconds, all other times are in microseconds. Numbers in italics represent 90% confidence intervals.

Test	Elap. 90%	Compute 90%	Read 90%
Cold	172.0 <i>0.84</i>	82.7 <i>0.12</i>	1890 <i>19.92</i>
Hot	131.5 <i>0.12</i>	81.4 <i>0.06</i>	782 <i>2.91</i>

Table 6: Read event count summary for the glimpse benchmark. Counts are the number of events that fell in that category averaged across the last 20 runs of the each test.

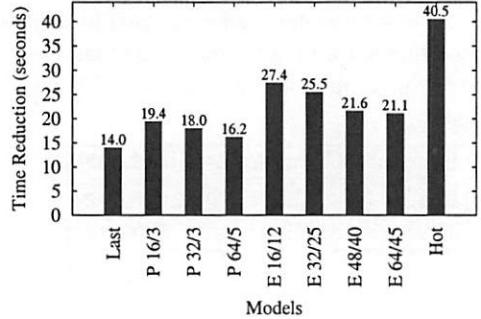
Test	calls	hits	partial	misses
Cold	24901	258	12828	11813
Hot	24901	5943	12819	6138

4.5 Patching and Building SSH

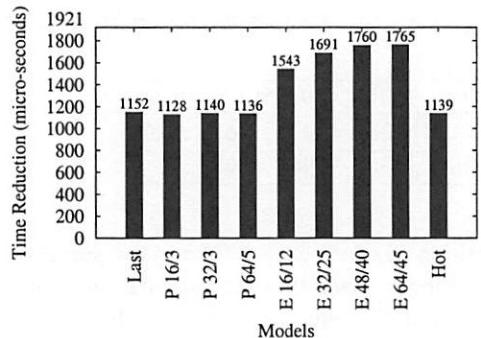
For our last benchmark we used the package **SSH**, versions 1.2.18 through version 1.2.31 to represent the compile and edit cycle. Thus the system is able to train on the initial version but needs to perform on subsequently modified versions of the source code. This benchmark represents our largest workload in that it consists of over 44,000 read events. However, a good percentage of these requests are already satisfied from the I/O caches. Here again we see results similar to those of the Gnu ld and Glimpse benchmarks. Total elapsed time was reduced by 11%, total I/O latency was reduced by 84% and read latencies were reduced by 70%.

We created the SSH benchmark to represent a typical compile and edit process. It addresses the concern that our other three benchmarks were being tested on a repeating sequence of the same patterns that it was trained on. This benchmark consists of compiling version 1.2.18 of the SSH package. Then the code base is patched to become 1.2.19 and recompiled. This process is iterated until version 1.2.31 is built. The result is a benchmark that provides a set of access patterns that change in a manner typical of a common software package.

Our models are trained on three compiles of version 1.2.18. We test predictive prefetching on a workload that patches the source to the next version and then compiles the new source code. This patching and build is repeated through the building of version 1.2.31. Because we are changing the source code with the var-



(a) Elapsed Time Reduction



(b) Read Latency Reduction

Figure 10: Reductions in elapsed times and read latencies for the Glimpse benchmark with the last successor, PCM, EPCM and hot cache tests. Bars marked with P and E represent PCM and EPCM tests respectively. Partition sizes (ps) and model order (mo) are labeled as ps/mo.

ious patches the patterns that result from the building represent a more realistic sequence of changing access patterns. This benchmark represents a case where our model may learn from the first build but will have to apply its predictions to a changing workload.

Tables 7 and 8 show the summary statistics for our SSH benchmark's workload. This workload has a CPU utilization of 89%. We observed a miss ratio of 0.12. The workload here represents that of a compile, edit and recompile process.

Figure 11 shows the results for our SSH benchmark. These results are consistent with those for our three previous benchmarks. Total elapsed time is reduced by 11%, while the I/O latency has been reduced by 84% and read latency has been reduced by 70%.

Table 7: Workload time summary for the SSH benchmark. Elapsed times are in seconds, all other times are in microseconds. Numbers in italics represent 90% confidence intervals.

Test	Elap. 90%	Compute 90%	Read 90%
Cold	302.0 <i>1.13</i>	263.6 <i>.82</i>	2813 <i>19.92</i>
Hot	268.4 <i>1.03</i>	262.8 <i>0.04</i>	861 <i>2.19</i>

Table 8: Read event count summary for the SSH benchmark. Counts are the number of events that fell in that category averaged across the last 20 runs of the each test.

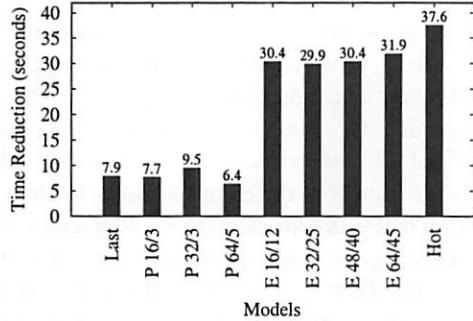
Test	calls	hits	partial	misses
Cold	44805	29552	13971	11282
Hot	44805	40839	13966	0

4.6 Training with Multiple Patterns

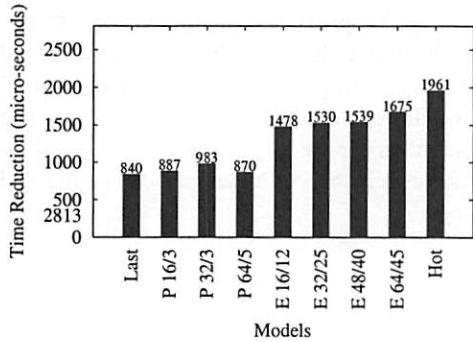
To briefly examine how predictive prefetching would work in an environment with multiple processes presenting different patterns, we ran two additional tests. The first tests trained on all four benchmarks, and then measured runs of the glimpse benchmark. The second test modified the Gnu ld benchmark to interject a string search over the source code in between runs of the benchmark. For these test we had a partition size of 64 and a maximum order of 45. These tests showed that the addition of other active patterns would have a modest effect on the performance of predictive prefetching.

For our first test we trained the system on 10 runs of all four benchmarks. Then we measured the performance of 20 runs of the glimpse benchmark. This test reduced elapsed time by 20.1 seconds while in §4.4 tests with same parameters reduced elapsed time by 21.1 seconds. We observed a read latency reduction of 1501 microseconds for this test, which is 69 microseconds less than that in §4.4.

While the above test examines multiple patterns there is little overlap in the files being accessed. For our second test we modified the Gnu ld benchmark to interject a second pattern of accesses in between each run of the benchmark. This second pattern is a recursive search of the files in the source code looking for the string ELVIS. For this test we saw the average elapsed time reduced by 3.0 seconds, which is 0.9 seconds less than we observed in §4.3. The read latencies were reduced 901 microseconds, 53 microseconds less than in §4.3.



(a) Elapsed Time Reduction



(b) Read Latency Reduction

Figure 11: Reductions in elapsed times and read latencies for the SSH benchmark with the last successor, PCM, EPCM and hot cache tests. Bars marked with P and E represent PCM and EPCM tests respectively. Partition sizes (ps) and model order (mo) are labeled as ps/mo.

From these modified tests we observe that the addition of other patterns into the training will have some modest effect on the performance of predictive prefetching.

4.7 Analysis of Results

Across the four different benchmarks we see somewhat similar results, significant reductions in total I/O latency and read latency with modest reductions in total elapsed time. From §4.2.2 we see that the computational overhead from our model and prefetch engine is negligible. A more detailed analysis of the overhead in predictive prefetching is available in previous work [15]. In comparing the predictive modeling techniques, EPCM seems to consistently outperform PCM and last successor. In comparing the different parameters for EPCM there doesn't seem to be a clear case for any specific settings.

To understand these results one should remember that the benchmarks presented here are—just as most other benchmarks—clean room simulations that attempt to recreate what occurs on a typical computer system. They should be considered in conjunction with our previous analysis of actual file system traces [13]. This work used long term traces from four different machines to show that the one trait that was consistent across all traces was predictable repeating patterns; specifically we saw that PCM could predict the next file access with an accuracy of 82%. This previous work indicates that the repetitive nature of our benchmarks is similar to the patterns that would be seen in a realistic workload. From these benchmarks we can see that predictive prefetching has the potential to significantly reduce total I/O latencies and read latencies, while providing modest improvements in total execution time. In real life the reduction one sees will be highly dependent on the specific characteristics of a their workload, such as how much I/O latency can be masked by prefetching.

5 Related Work

The use of compression modeling techniques to track access patterns and prefetch data was first examined by Vitter, Krishnan and Curewitz [25]. They proved that for a Markov source such techniques converge to an optimal on-line algorithm, and then tested this work for memory access patterns in an object-oriented database and a CAD System. Chen *et al.* [8] examine the use of *FMOCM* type models for use in branch prediction. Griffioen and Appleton [6] were the first to propose a *graph-based* model that has seen use across several other applications [23, 21]. Lei and Duchamp [18] have pursued modifying a UNIX file system to monitor a process' use of fork and exec to build a tree that represents the processes execution and access patterns. Kuennen *et al.* [17] have developed the concept of a *semantic distance* and used this to drive an automated hoarding system to keep files on the local disks of mobile computers. Madhyastha *et al.* [19] used hidden Markov models and neural networks to classify I/O access patterns within a file.

Several researchers are exploring methods for cache resource management given application-provided hints. Patterson *et al.* [24] present an *informed prefetching* model that applies cost-benefit analysis to allocate resources. Cao *et al.* [4] examine caching and prefetching in combination and present four rules for successfully combining the two techniques and evaluate several

prefetching algorithms including an *aggressive prefetch* algorithm. Kimbrel *et al.* [10] present an algorithm that has the advantages of both *informed prefetching* and *aggressive prefetch* while avoiding their limitations.

6 Future Work

While this work has shown that file reference patterns provide valuable information for caching, and use of such information can greatly reduce I/O latency, we have also found certain areas that require further study. We hope to examine the following issues.

The paging of predictive data to and from disk is critical to the success of predictive prefetching. While our implementation was done in a manner that facilitates such functionality, we have not directly addressed this issue.

The idea of *partition jumping* would use multiple partitions to continue in a sequence past the end of one partition and into another partition that begins with the last n symbols of the sequences. This would allow EPCM to make predictions deeper than the partition size. This new method would generate predictions with EPCM as before, but when a descendent with no children was found, the last n symbols in the pattern would be used as an n -order context into a new partition from which predictions would continue. This would enable EPCM to look into other partitions once it has reached the end of the current partition, and enable smaller partitions to predict further ahead than their partition size would normally allow.

In our test environment, we ran the same benchmark test consistently, so our models saw no variation. As a result, they generated no erroneous prefetching. It would be instructive to use trace-based simulations to investigate how often our models would incorrectly prefetch. If we then forced an implementation to make this percentage of incorrect prefetches, we could gauge the impact of incorrect prefetching on the system as a whole.

7 Conclusions

Comparing the predictive models, the last successor and PCM models saw reasonable improvements, but the increased lead time of EPCM's prefetching enables significantly greater improvements in read latencies and

in total elapsed times. Although last successor based prefetching can be effective in reducing I/O latencies, its limitations are that it cannot predict more than the next event and it provides no confidence estimates for the predictions. While PCM based prefetching provides a measure of likelihood with each prediction, this method cannot predict more than the next event, limiting its ability to reduce I/O latencies. We have seen that EPCM based prefetching can greatly reduce I/O latencies by predicting further ahead than PCM based prefetching.

While these tests have clearly shown that predictive prefetching can greatly reduce I/O latencies, we note that these tests are limited representations of any common computer workload, and that several key issues still need to be addressed to implement a system that could be widely used. These tests run the same benchmark repeating the same patterns. While our simulations with file system traces clearly show a strong degree of correlation between file access events, our repetition of the same benchmark numerous times has artificially increased this correlation. Additionally this experimental implementation does not store any of the predictive data to disk. We envision that this would affect our system by slightly increasing the I/O activity and significantly decreasing memory overhead. How this would affect performance is unclear; we have not studied the effects such changes would have on performance, and can only speculate based on experiences with this implementation. Any practical implementation of predictive prefetching would need to handle these issues.

These results show that predictive prefetching can significantly reduce I/O latencies and shows useful reductions in total runtime of our benchmarks. Our prototype focused on using predictive prefetching to reduce the latencies of read system calls, and this is exactly what we have seen. From the reductions in elapsed time, we have shown that a predictive prefetching system as a whole offers potential for valuable performance improvements. In the best case, such a system performs almost as well as when all of the data is already available in RAM. However, care must be taken in the design of a predictive prefetching system to ensure that the prefetching uses resources wisely and does not hinder demand driven requests. Nevertheless these test have shown that, for the workloads studied, predictive prefetching has the potential to remove significant portions of I/O latencies.

8 Acknowledgments

The authors are grateful to the many people that have helped our work. Ahmed Amer, Randal Burns, Scott Brandt and the other members of the Computer Systems Lab provided useful comments and support. The Usenix Association, National Science Foundation and the Office of Naval Research have provided funding that supported this work. The Linux community was helpful in working with the Linux kernel. Rod Van Meter and Melanie Fulgham provided helpful comments on early drafts of this work. Nokia's Clustered IP Solutions supported Dr. Kroeger's efforts in bringing this work to publication.

References

- [1] M. Beck, H. Bohme, M. Dziadzka, U. Kunitz, R. Magnus, and H. Bohme, *Linux Kernel Internals*. Addison-Wesley Publishing Company, 1997.
- [2] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.
- [3] A. Bestavros, "Speculative Data Dissemination and Service to Reduce Server Load, Network Traffic and Service Time for Distributed Information Systems," in *Proceedings of the 1996 International Conference on Data Engineering*, (New Orleans, Louisiana), Mar 1996.
- [4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li, "A study of integrated prefetching and caching strategies," in *Proceedings of the 1995 SIGMETRICS Conference*, pp. 188–197, ACM, May 1995.
- [5] F. Chang and G. A. Gibson, "Automatic I/O hint generation through speculative execution," in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation*, pp. 1–14, 1999.
- [6] J. Griffioen and R. Appleton, "Performance measurements of automatic prefetching," in *Proceedings of the 1995 Parallel and Distributed Computing Systems Conference*, pp. 165–170, IEEE, September 1995.
- [7] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *Transactions on Computer Systems*, vol. 6, pp. 51–81, February 1988.

- [8] T. N. M. I-Cheng K. Chen, John T. Coffey, "Analysis of branch prediction via data compression," in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 128–13, SIGOPS, ACM, October 1996.
- [9] J. Katcher, "Postmark: A new file system benchmark," tech. rep., Network Appliance Inc., 2000.
- [10] T. Kimbrel, A. Tomkins, R. H. Patterson, B. Bereshad, P. Cao, E. W. Felton, G. A. Gibson, A. Karlin, and K. Li, "A trace-driven comparison of algorithms for parallel prefetching and caching," in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, pp. 19–34, USENIX, October 1996.
- [11] D. E. Knuth, *Sorting and Searching*, vol. 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, 1973.
- [12] T. M. Kroeger and D. D. E. Long, "Predicting file-system actions from prior events," in *Proceedings of the USENIX 1996 Annual Technical Conference*, pp. 319–328, USENIX, January 1996.
- [13] T. M. Kroeger and D. D. E. Long, "The case for efficient file access pattern modeling," in *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, IEEE, March 1999.
- [14] T. M. Kroeger, "Predicting file system actions from reference patterns," Master's thesis, University of California Santa Cruz, March 1997.
- [15] T. M. Kroeger, *Modeling File Access Patterns to Improve Caching Performance*. PhD thesis, University of California Santa Cruz, March 2000.
- [16] G. Kuennig, G. J. Popek, and P. Reiher, "An analysis of trace data for predictive file caching in mobile computing," in *Proceedings of the USENIX Summer Technical Conference*, pp. 291–303, USENIX, 1994.
- [17] G. H. Kuennig and G. J. Popek, "Automated hoarding for mobile computers," in *Proceedings of the Sixteenth Symposium on Operating Systems Principles (SOSP-97)*, vol. 31,5, (New York), pp. 264–275, ACM Press, Oct. 5–8 1997.
- [18] H. Lei and D. Duchamp, "An analytical approach to file prefetching," in *Proceedings of the USENIX 1997 Annual Technical Conference*, pp. 275–288, USENIX, January 1997.
- [19] T. Madhyastha and D. A. Reed, "Input/output access pattern classification using hidden Markov models," in *Proceedings of the Fifth Workshop on Input/Output in Parallel and Distributed Systems*, pp. 57–67, ACM, Nov 1997.
- [20] U. Manber and S. Wu, "GLIMPSE: A tool to search through entire file systems," in *Proceedings of the USENIX Winter Technical Conference*, (Berkeley, CA, USA), pp. 23–32, USENIX, Winter 1994.
- [21] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. E. Anderson, "Improving the performance of log-structured file systems with adaptive methods," in *Proceedings of the Sixteenth Symposium on Operating Systems Principles (SOSP-97)*, vol. 31,5 of *Operating Systems Review*, (New York), pp. 238–251, ACM, Oct. 5–8 1997.
- [22] J. Ousterhout, "Why aren't operating systems getting faster as fast as hardware?," in *Proceedings of the USENIX Summer Technical Conference*, pp. 247–56, USENIX, June 1990.
- [23] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve world wide web latency," in *Proceedings of the 1996 SIGCOMM Conference*, pp. 25–35, ACM, July 1996.
- [24] H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka, "Transparent informed prefetching," in *Proceedings of the Fifteenth Symposium on Operating Systems Principles (SOSP-95)*, pp. 21–34, ACM, December 1995.
- [25] J. S. Vitter and P. Krishnan, "Optimal prefetching via data compression," *Journal of the ACM*, vol. 43, pp. 771–793, September 1996.
- [26] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. IT-24, pp. 530–6, September 1978.

Extending Heterogeneity to RAID level 5*

T. Cortes and J. Labarta

Departament d'Arquitectura de Computadors

Universitat Politècnica de Catalunya

{toni, jesus}@ac.upc.es, <http://www.ac.upc.es/hpc>

Abstract

RAIDs level 5 are one of the most widely used kind of disk array, but their usage has some limitations because all the disks in the array have to be equal. Nowadays, assuming a homogeneous set of disks to build an array is becoming a not very realistic assumption in many environments, especially in low-cost clusters of workstations. It is difficult to find a disk with the same characteristics as the ones in the array and replacing or adding new disks breaks the homogeneity. In this paper, we propose a block-distribution algorithm that can be used to build disk arrays from a heterogeneous set of disks. We also show that arrays using this algorithm are able to serve many more disk requests per second than when blocks are distributed assuming that all disks have the lowest common speed, which is the solution currently being used.

1 Introduction

Heterogeneous disk arrays are becoming (or will be in a near future) a common configuration in many sites. Let us describe two scenarios that end up in a heterogeneous disk array. The first one appears whenever a component of a traditional array fails and it has to be replaced by a new one. As disk technology improves quite rapidly, it is quite probable for the new disk to be faster and larger than the ones already in the array [7]. A similar scenario appears when the capacity needs of a site grow and new disks have to be acquired to grow the size of the array (by increasing the number of disks in the array). In this case, it will also be difficult to buy the same disks as the ones in the original configuration, and thus newer disks will be added. In

both cases, we will make the array a heterogeneous one because it will be made of disks with different characteristics. This kind of situation is especially common in low-cost clusters of workstations, where cost is an important issue and old components have to be used as well as possible. According to the study performed by Dr. Grochowski at IBM [7], disk capacity nearly doubles every year while the price per Mbyte is decreasing about 40% per year. This means that the price of arrays will remain about the same throughout the years, although the capacity will be increased a lot, of course. If a given site wants to buy a 32 disk array (assuming for example 18GB Seagate disks at today prices), it costs between \$17000 and \$26400 (depending on the interface, RPM, and seek time) [18]. At this price, changing all these disks at a time because one of them breaks is too expensive for many institutions and/or companies, especially if the problem can be solved by just buying a single disk. The only exception appears when the site does not need to grow its capacity and thus replacing the 32-disk array by a few new ones (reducing the size of the array) is enough. Nevertheless, this does not seem to be the trend as disk usage grows constantly.

To handle this kind of disk array, current systems do not take into account the differences between the disks. All disks are treated as if they had same capacity (the smallest one) and performance (the slowest one). This is not the best approach because improvements in both capacity and response time of the heterogeneous array could be achieved if each disk were used accordingly to its characteristics.

In this work, we present a simple solution to this problem by proposing AdaptRaid5, a block-distribution algorithm that improves the performance and effective capacity of heterogeneous disk arrays compared to current solutions. We should note that this proposal has been especially evaluated for scientific and general purpose workloads (under-

*This work has been supported by the Spanish Ministry of Education (CICYT) under the TIC-95-0429 contract.

standing as workload the requests that reach the disk controller, after being filtered by the file system cache) because the multimedia case has already been addressed quite successfully by other research groups [6, 17, 24]. Nevertheless, the proposed algorithm also works well in a multimedia environment.

This paper is divided into 8 Sections. Section 2 presents the most relevant work in the area of heterogeneous disk arrays. Section 3 introduces the reader to some important concepts that need to be clarified before describing the algorithm, which is explained in full detail in Section 4. Section 5 presents the methodology used to obtain the results presented in Section 6. Section 7 presents the future work we plan to do in this field. Finally, Sections 8 and 9 present the conclusions that can be extracted from this work and how to get more information about this work.

2 Related Work

Some projects have already addressed the same problem, but they have been focused on multimedia systems (and especially video and audio servers). The work done by Santos and Muntz [17] proposed a random distribution with replications to improve the short and long-term load balance. In a similar line, Zimmermann proposed a data placement policy based on the creation of logical disks composed of fractions or combinations of several physical disks [24]. Finally, Dan and Sitaran proposed the usage of fast disks to place "hot" data while the less important data would be located in the slow disks [6]. The main difference from our approach is that all these projects were targeted to multimedia systems while we want a solution for general purpose and scientific environments. Due to their focus on multimedia, they could make some assumptions such as that very large disk blocks (1Mbyte) are used, that reads are much more important than writes and that the main objective is to obtain a sustained bandwidth as opposed to achieving the best possible response time. These assumptions are not valid in our environment where blocks are only a few Kbytes in size, writes are as important as reads, and sustained bandwidth is not as important as the fastest response time. We have to keep in mind that we evaluate the accesses that reaches the disk controller after being filtered by the file-system cache.

The only two works, as far as we know, that deal with this problem in a non-multimedia environment are the HP-AutoRaid [23] and a software RAID that has been implemented in Linux [21]. In the case of the AutoRaid, heterogeneity in the devices is not the objective, but its architecture supports different kind of disks. Nevertheless, in that work only size has been taken into account and no studies to improve performance by using the disks according to their characteristics have been presented. In the software RAID in Linux, any of the disks in the array can be built by putting several disks together. Each disk will store part of the blocks assigned to this *virtual* disk. The problem with this approach is that it is too simple because it only works if you can find a set of disks that match the size of the others in the array (unless you want to waste disk space). Furthermore, it only works for RAIDs level 0, and not for level 5.

Other projects have also dealt with a heterogeneous set of disks, but their objective was to propose new architectures using different disks for different tasks. Along this line we could mention the DCD architecture [10]. In our work, we do not try to decide which is the best hardware and then buy it, we want to deal with already existing devices whichever they are.

The work done by Holland and Gibson in 1992 [9] and by Lee and Katz in 1993 [12] is also related to this project, although not from the heterogeneity point of view. In both studies, ways to handle stripes with smaller striping units than disks in the array are presented. This idea is also used in our work, as will be seen throughout the paper.

Finally, our research group has also proposed a solution to the same problem for disk arrays level 0 [5]. Although it is a much simpler algorithm, because there are no parity problems, many of the ideas presented here have evolved from that first proposal.

3 Preliminary Issues

3.1 Disk Arrays and Parallelism

Disk arrays were especially designed to group several disks into a single address space and to offer high bandwidth by exploiting data access par-

allelism. Understanding how this parallelism improves the performance of an array is very important to understanding the design and results presented in this paper.

A first kind of parallelism is achieved within a single request. In this case, all disks work together to fulfill a single request and thus the time spent transferring data from the magnetic surface is divided by the number of disks.

A second kind of parallelism occurs when several requests do not use all disks in the array and can be served in parallel. This kind of parallelism makes sense when requests are small compared to the size of the stripe. If requests are large, they will use all disks and the parallelism between requests will decrease significantly.

3.2 Small-Write Problem

One of the most important performance problems in a RAID5 is the small-write problem. In this kind of array, writing data implies that the parity information has to be updated. For this reason, it is recommended to write full stripes as the parity can be computed only using the blocks to be written. If a write operation does not write all the blocks in a stripe, some blocks have to be read from the array to recompute the new parity. This means that a write also implies a read, which penalizes the performance of the operation.

In this work, we consider the read-write-modify approach as opposed to the regenerate-write [3] because it offers more parallelism between requests. The first one (read-write-modify) consists of reading the same blocks that are being written and the parity block. Then, the parity block is XORed with the old blocks (just read) and with the new blocks (just to be written) obtaining the new parity block. The other possibility (regenerate-write) is to read the blocks that are not being modified and thus the new parity blocks can be computed because we have all the blocks in the stripe.

4 AdaptRaid5

4.1 Block-Distribution Algorithm

The best way to understand this algorithm is to describe its evolution starting from the most intuitive, but problematic, version. Then, we discuss the problems we have detected and the solutions we have proposed. To conclude, we present the final version, which should produce a high-performance and high-capacity heterogeneous RAID5.

Intuitive idea

As we have already mentioned, replacing an old disk by a new one or adding new disks to an old array are two common scenarios. In both cases, new disks are usually larger and faster than the old ones [7]. For this reason, we start by assuming that faster disks are also larger, although we will drop this assumption at the end of this section.

The intuitive idea is to place more data blocks in the larger disks than in smaller ones. This makes sense when larger disks are also faster, and thus they can serve more blocks per unit of time. Following this idea, we propose to use all D disks (as in a regular RAID5) for as many stripes as blocks can fit in the smallest disk. Once the smallest disk is full, we use the rest of the disks as if we had a disk array with $D-1$ disks. This distribution continues until all disks are full with data.

A side effect of this distribution is that the system may have stripes with different lengths. For instance, if the array has D disks where F of them are fast, the array will have stripes with $D-1$ blocks (plus the parity block), but it will also have stripes with $F-1$ blocks (plus the parity block). This was not a problem in RAIDs level 0 [5], nevertheless, the effect it may have on a RAID 5 will be discussed later in this paper.

Finally, the parity block for each stripe is placed in the same position it would have been in a regular array with as many disks as blocks in the stripe.

In Figure 1, we present the distribution of blocks in a five-disk array where disks have different capacities. Each block has been labeled with the block number in the array followed by the stripe in which it is

		Disk	0	1	2	3	4
		Stripe	0-0	1-0	2-0	3-0	P0
		1	4-1	5-1	6-1	P1	7-1
		2	8-2	9-2	P2	10-2	
		3	11-3	P3	12-3	13-3	
		4	P4	14-4	15-4	16-4	
		5	17-5	18-5	19-5	P5	

Figure 1: Distribution of data and parity blocks according to the intuitive version.

located (i.e. 8-2 represents data block 8, which is in the stripe number 2). Parity blocks are just labeled with a P and the stripe to which they belong. We have to notice that the last block of the largest disk is not used. This happens because stripes must be at least two blocks long, otherwise there is no room to store the parity block for the stripe.

Reducing the small-write problem

As we mentioned in Section 3.2, the file system or controller should organize writes in order to avoid small writes as much as possible [1, 8, 20]. On the other hand, our array has stripes with different sizes and thus if the file system or controller optimizes writes for a given stripe size, it will not be appropriate for stripes with a different size. For instance, if the file system tries to write chunks of 3 blocks (plus the parity one) in a 4-disk stripe, a full stripe will be written. However, if the same chunk is written into a 3-disk stripe, it will perform one full write for two of the data blocks and a small write for the other data block. This means that the performance of a write operation will greatly depend on the stripe it is written to.

The solution to this problem can be approached from two different levels: file system or device. In the first case, the file system has to know that there are different stripe sizes in order to optimize writes accordingly. In the second case, which is the one we propose, the array hides the problem from the file system that assumes a fixed stripe size.

The array can hide the problem of having different stripe sizes by making sure that the number of data blocks in each stripe is a divisor of the number of

		Disk	0	1	2	3	4
		Stripe	0-0	1-0	2-0	3-0	P0
		1	4-1	5-1	6-1	P1	7-1
		2	8-2	9-2	P2		
		3	10-3	P3	11-3		
		4	P4	12-4	13-4		
		5	14-5	15-5	P5		

Figure 2: Distribution of data and parity blocks when the stripe size is taken into account.

data blocks in the largest stripe, which we assume is what is being used by the file system. This condition guarantees that full stripes, from the file system point of view, are divided into a set of full stripes, and thus the number of small writes is not increased.

In Figure 2, we present the new distribution for the example in Figure 1. We should notice that the last four blocks in disk 3 become unused. As we have mentioned, the number of data blocks in a stripe has to be a divisor of the data blocks in the largest one. In this example, the largest stripe has 4 data blocks, and thus a stripe with three data blocks is not a valid one. For this reason, stripe number 2 becomes a three-block stripe and all the space in disk 3 that comes after P1 remains unused.

Increasing the effective capacity

The distribution for solving the small write problem above has created a capacity problem in that some blocks must go unused to keep the smaller stripe sizes divisible into the maximum stripe size. For example, the dark blocks in Figure 2 cause the utilization of disk 3 to be only 33%. Thus, the next step is to reclaim our ability to utilize those extra blocks.

We will describe this optimization in two steps. First, we will find a way to use all the available disks without worrying about the capacity. And second, we will use this distribution to increase the effective capacity.

The first problem, then, is how to map stripes that are N -blocks long in a set of D disks ($D > N$) using all the disks. One way to do this mapping is to

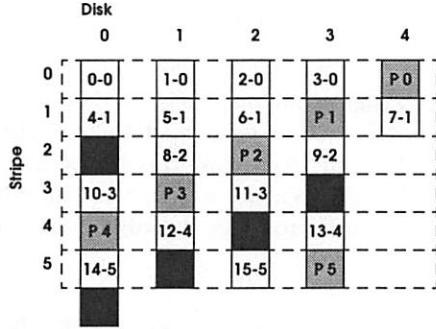


Figure 3: Distribution of stripes, which are 3-blocks long, among four disks.

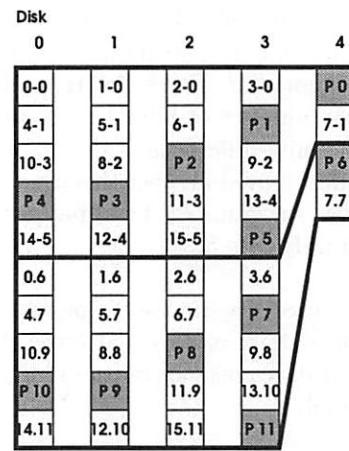


Figure 5: Example of pattern repetition.

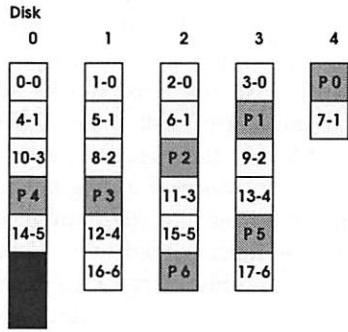


Figure 4: Distribution of stripes, which are 3-blocks long, among four disks filling all empty blocks.

start each stripe in a different disk. For instance, if stripe i starts in disk d , then stripe $i+1$ should start on disk $d-1$. Figure 3 shows an example where stripes that are 3-blocks long (2 data plus 1 parity) are distributed among four disks. Please notice that this only happens for stripes 2 to 5.

The previous step uses all disks, but the number of unused blocks is not reduced at all. To fill these unused blocks we can use a *Tetris-like* algorithm. We can push all blocks so that all empty spaces are filled. Figure 4 presents the previous example once the *pushing* has been done. We can observe now, that all the blocks in the disk are used regardless of the size of the stripe (2 additional data blocks plus one parity block can be accommodated). With this algorithm, we can have stripes with different sizes while all the blocks in all disks are used to store either data or parity information.

Reducing the variance in parallelism

If we apply the algorithm as we have presented it so far, we observe that longer stripes are placed in the lower portion of the address space of the array while the shorter ones appear in the higher portion of the address space. This means that requests that fall in the lower part of the address space can use more disks (longer stripes) while the requests that fall in the higher part of the address space only use a small subset of the disks (shorter stripes).

This can be a problem if our file system tries to place all the blocks of a file together, which is a common practice [13, 14, 19]. This means that a given file may have most of its blocks in the lower part of the address space (long stripes) while another file may have all its blocks in the higher part of the address space (short stripes). Although the global access in the system will be an average, the first file will have a faster access time (more parallelism) while the second one will have a slower access time (less parallelism). For this reason, evenly distributing the location of long and short stripes all over the array will reduce the variance between the accesses in the different portions of the disk array, which we believe is how the storage system should behave.

To make this distribution, we introduce the concept of a pattern of stripes. The algorithm assumes, for a moment, that disks are smaller than they actually are (but with the same proportions in size) and distributes the blocks in this *smaller* array. This distribution becomes the pattern that is repeated

until all disks are full. The resulting distribution has the same number of stripes as the previous version of the algorithm. Furthermore, each disk also has the same number of blocks as in the previous version. The only difference is that short and long stripes are distributed all over the array, which was our objective. An example of this pattern repetition can be seen in Figure 5.

With this solution, we can see the pictures presented so far (Figures 1, 2, or 4) as patterns that can be repeated in disks thousands of times larger than the ones presented.

It is also important to notice that the concept of patterns will simplify the algorithm to find a block as will be described later (Section 4.2).

Limiting the size of the pattern

Finally, we want to solve a very focused problem that will only appear in special cases, but that may be important in some cases. Nevertheless, the solution is very simple and has no negative side effects in the rest of cases, making it appropriate to be implemented.

In a regular disk array, all stripes are aligned to a multiple of the number of data blocks in the stripe. We may have systems, or applications, that try to align their full-stripe requests to the beginning of a stripe to avoid making extra read operations. For example, if we have a distribution where the pattern is the one in Figure 4, accessing 4 blocks starting from block 16 should be a full stripe. However, it is not with our new block-distribution algorithm.

Before presenting the solution, we need to define the concept of *reference stripe*. A reference stripe is the stripe that the system or application assumes to be a full stripe. For instance, in the previous example the reference stripe has 4 data blocks and 1 parity block.

The solution to this problem is quite simple. The algorithm only has to make sure that the number of data blocks in a pattern is a multiple of the number of blocks in the reference stripe. This condition guarantees that all repetitions of the pattern start at the beginning of a file system *full stripe*. The result of applying this last step in the example can be seen in Figure 5.

Generalizing the solution

So far, our algorithm has been based on an assumption that the size of disks and their performance grow at the same pace, but this is not usually the case [7]. For this reason, we want to generalize the algorithm in order to make it usable in any environment.

If we examine the algorithm we can see that there are two main ideas that can be parameterized. The first one is the number of blocks we place in each disk. So far, we assumed that all blocks in a disk are to be used. Now, we want to add a parameter to the algorithm that defines the proportion of blocks that are placed in each disk. The **utilization factor (UF)**, which is defined on a per-disk basis, is a number between 0 and 1 that defines the relationship between the number of blocks placed in each disk. The disk with the most blocks always has a UF of 1 and the rest of disks have a UF related to the number of blocks they use compared to the most loaded one. For instance, if a disk has a UF of 0.5, it means that it stores half the number of blocks as compared to the most loaded one. This parameter allows the system administrator to decide the load of the disks. We can set values that reflect the size of the disks, or we can find values that reflect the performance of the disk instead of the capacity.

The second parameter is the number of **stripes in the pattern (SIP)**. The number of stripes in the pattern indicates how well distributed are the different kinds of stripes along the array. Nevertheless, we should keep in mind that smaller disks will participate in less than SIP stripes.

Figure 5 presents a graphic example of how blocks are distributed in the first two repetitions of the pattern if we use the following parameters: $UF_0 = UF_1 = UF_2 = UF_3 = 1$, $UF_4 = 0.4$ and $SIP = 6$. Please note that there are no empty blocks in the picture because we assume much larger disks and the empty blocks would be placed at the end. Remember that the picture only shows the first two repetitions of the pattern.

Fast but small disks: a special case

The current algorithm can be used with any kind of disks. Nevertheless, it does not make much sense if the fastest disk is also significantly smaller. In this

case, a better use for these disks would be to keep “hot data” as proposed by Dan and Sitaran [6].

4.2 Computing the Location of a Block

Besides all the aspects already mentioned about performance of disk accesses, we also need to make sure that finding the physical location of a given block can be done efficiently.

This is done in a very simple way. When the system boots, the distribution of blocks in a pattern is computed and kept in three tables. The first one (**location**) contains the disk and position within that disk of any block in the pattern. The second one (**parity**) keeps the location of the parity block for each stripe. Finally, the third table (**Blks_per_disk_in_pattern**) stores the number of blocks each disk has in a pattern. These tables should not be too large. In our experiments the **position** table has 152 entries, the **parity** one only has 19 entries, and the **Blks_per_disk_in_pattern** has 9 entries. These sizes can be assumed by any RAID controller or file system. The formulas to compute the location of a given block (B) follow:

```
disk(B) = location[B%Blks_in_a_pattern].disk
pos(B) = location[B%Blks_in_a_pattern].pos +
          (B/Blks_in_a_pattern) * Blks_per_disk_in_pattern[disk(B)]
```

As these operations are very simple, the algorithm to locate blocks is very fast. To check this time, we profiled the simulator and we found that the time spent in deciding the location of blocks was less than $81\mu s$ in average per request¹, which is insignificant compared to the time of a disk access.

All tests presented in this paper were performed simulating an array with a combination of slow and fast disks. The model used for these disks is the one proposed by Ruemmller and Wilkes [16]. The parameters used for the slow disks were taken from the Seagate Barracuda 4LP [18] and to emulate the fast disk we used the parameters of a Cheetah 4LP [18], which is also a Seagate disk. A list with some important characteristics for each disk (controller and drive) are presented in Table 1. Finally, the size used for the striping unit is 128Kbytes. This size has been computed using the ideas presented by Chen et al. [2]. Although the formulas presented in that paper were for homogeneous disk arrays, we have assumed they would be adequate for heterogeneous ones.

Table 1: Disk characteristics.

	Fast Disk	Slow Disk
Size		
Disk size	4.339 Gb	2.061 Gb
Cache size	512Kbytes	128Kbytes
Sector size	512Bytes	512Bytes
Cache model		
Read/Write fence	64Kbytes	64Kbytes
Prefetching	YES	YES
Immediate report	YES	YES
Overheads		
New-command	$1100\mu s$	$1100\mu s$
Track switch	$800\mu s$	$800\mu s$
Bandwidth		
RPM	10033	7200
Seek model		
Limit (in cylinders)	600	600
Sort: $a+b*\sqrt{d}$ μs	$a = 1.55$ $b = 0.155134$	$a = 3.0$ $b = 0.232702$
Long: $a+b*(d)$ μs	$a = 4.2458$ $b = 0.001740$	$a = 7.2814$ $b = 0.002364$

5 Methodology

5.1 Simulation and Environment Issues

In order to perform this work, we have used HRaid [4], which is a storage-system simulator². The simulator has been validated using the HP-92 suit of traces [15, 16] and also comparing the results of many tests to the ones obtained by D. Kotz’s simulator [11], which is also a validated simulator.

¹Times taken in a SGI2000

²<http://www.ac.upc.es/homes/toni/software.html>

These disks and the hosts were connected through a Gigabit network ($10\mu s$ latency and 1Gbits/s bandwidth). We simulated the contention of the network, but no protocol overhead was simulated.

We also have to keep in mind that in the simulations we only took the network and disks (controller and drive) into account. The possible overhead of the requesting hosts was not simulated because it greatly depends on the implementation of the file system. The only issue we simulated from the file system was that it can only handle 10 requests at a time. The rest of requests wait in a queue until one of the previous requests has been served.

Finally, we have to mention that when using the synthetic traces presented in the next section, we made 10 runs for each one of them (all with different seed to generate the access pattern) and report the average value. In these runs we always obtained very similar results and the difference was never larger than 2%.

5.2 Workload issues

In order to get the first results, we have studied the behavior of the system on a set of synthetic workloads based on the following parameters:

- **Kind of request:** whether requests were reads or writes.
- **Request size:** the size of all the requests in the load.
- **Request alignment:** the position of the requests is always chosen randomly, but the start of the request can be either aligned to a block in the first disk (to avoid small writes) or not.

Table 2 presents the characteristics of the synthetic workloads used.

Table 2: Synthetic-workload characteristics.

	Request Size	Aligned	Operation Type
W8	8Kbytes	No	Writes
W256	245Kbytes	No	Writes
W1024	1024Kbytes	Yes	Writes
W2048	2048Kbytes	Yes	Writes
R8	8Kbytes	No	Reads
R2048	2048Kbytes	Yes	Reads

On the other hand, we also wanted to obtain results for a real system, and thus we used a portion of the traces gathered by the Storage System Group at the HP Laboratories (Palo Alto) in 1999 [22]. These traces represent a detailed characterization of every low-level disk access generated in the system over a 6 month period. This system contained a file server and some workstations used by the people in the Storage System Group to perform their work (compilations, edition, databases, simulations, etc.). As the size of the traces was too large (6 months) we will only present the results obtained during the

busiest hour of February 14th. The tested portion has 159208 reads and 115044 writes and the average request size is around 12 Kbytes. With these traces, as with most traces, dependencies such as that a given operation has to follow another one are not recorded. However, this does not invalidate the results presented because the general load they represent continues to be real.

5.3 Configurations Studied

All the experiments presented in this paper have been done using disk arrays with 9 disks. This number of disks is large enough to see the possible advantage and limitations of the proposal. Furthermore, it is small enough to make things easy to understand.

Another important issue is the way small writes are handled. All the arrays we have evaluated used the read-write-modify algorithm, which means that the blocks read in a small write are the same ones as the blocks written [3]. This option has been used because it increases the parallelism between requests.

For simplicity, the configurations used always have all fast disks in the first positions and the slow ones in the last position of the array.

Finally, we have chosen a single SIP of 19 for all experiments, also for simplicity reasons. Regarding the utilization factors we have used a UF of 1 for the fast disks and .46 for the slow disks. These values have been decided experimentally and a sensitivity analysis for this parameter is presented in Section 6.6. We know that better values could be used for some of the experiments, but this is not the important issue as we want to prove the goodness of the idea and not to propose the best possible parameters.

5.4 Reference Systems

We have compared AdaptRaid5 with the following two base configurations:

- **RAID5:** This is the traditional RAID5 algorithm and it uses all the disks (fast and slow). It is important to notice that this leads to fast disks being treated as if they were slow ones

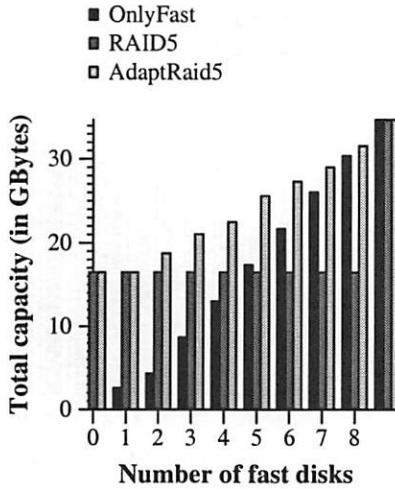


Figure 6: Effective capacity for the studied configurations.

and that only a portion of their capacity is effectively used.

- **OnlyFast:** This is also a traditional RAID5, but only using fast disks. The number of fast disks will be the same as the number of fast disks in the heterogeneous configuration. This comparison will give us the idea of whether it is better to throw the old disks away instead of using them.

6 Experimental Results

6.1 Capacity Evaluation

We present a graph (Figure 6) of the effective capacity based only on data blocks as we vary the number of fast disks out of the total of 9 disks for each distribution algorithm used. We can see that AdaptRaid5 is the one that obtains the largest capacity. This happens because it knows how to take advantage of the capacity of all disks in the array. Furthermore, we can see that the extra number of parity blocks used by our proposal does not affect the effective capacity significantly.

6.2 Full-Write Performance

The performance obtained by a RAID5 when a full stripe is written is one of the important results for

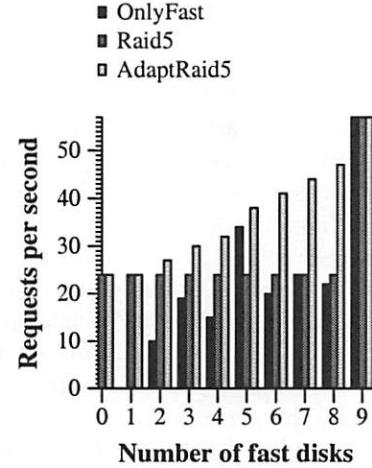


Figure 7: Writing 1024Kbytes blocks (W1024).

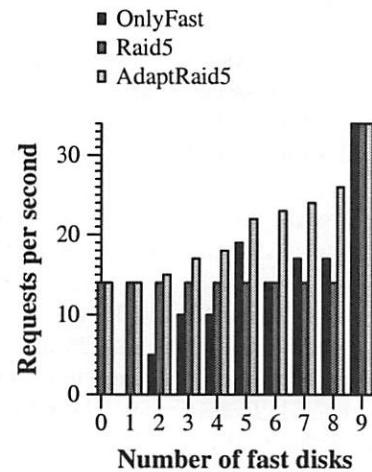


Figure 8: Writing 2048Kbytes blocks (W2048).

this kind of array. For this reason, we start evaluating the case where a write operation does not imply a previous read. To study this performance, we have measured the number of requests per second each of the evaluated systems can handle when requests are 1024Kbytes and 2048Kbytes long (workloads W1024 and W2048 described in Section 5.2). Although these may seem to be very large requests for the target environment, it is the only way to test full writes. Controllers or file systems may use logging and achieve such request sizes in non multimedia environments. Figures 7 and 8 present these results.

If we concentrate our attention on each of the systems individually, we can see that RAID5 does not change its performance when more of the disks are fast. This happens because this algorithm does not know how to use the better performance of newer

disks.

The second system, OnlyFast, has a very inconsistent behavior. It can achieve high performance under some configurations and a very bad one under others. The reason behind this behavior is the increase in the number of small writes. As we have mentioned in Section 4.1, if the number of data disks used is not a divisor of number data blocks in a stripe, a full-stripe write operation ends up performing a small write. This scenario occurs when the system has 4, 6, 7 and 8 disks. In the rest of the configurations, the performance obtained by OnlyFast is quite good and proportional to the number of fast disks. We should notice that this system has not been evaluated for 0 or 1 fast disks because we need at least 2 disks to build a RAID5.

The last evaluated system is our proposal (AdaptRaid5). We can observe that the performance of this system increases at a similar pace as the number of fast disks used, which was our objective.

If we compare the behavior of traditional RAID5 with our proposal, we can see that AdaptRaid5 always achieves a much better performance. This happens because AdaptRaid5 knows how to take advantage of fast disks while RAID5 does not. The only exception to this rule appears when only 0 or 1 fast disks are used. In this case, AdaptRaid5 cannot use the fast disks in any special way.

The comparison between AdaptRaid5 and OnlyFast also shows that our proposal is a better one. On the one hand, AdaptRaid5 is much more consistent than OnlyFast and it does not present a bad performance in any of the configurations. On the other hand, our system always obtains a better performance than OnlyFast. AdaptRaid5 is faster because it takes advantage of the parallelism within a request (it has more disks), which is very important when only a few fast disks are available or when requests are large. Furthermore, when OnlyFast starts to take advantage of the parallelism (when more fast disks are used), AdaptRaid5 starts to use the slow disks less frequently, which out-weights the improvements of OnlyFast.

6.3 Small-Write Performance

The other possibility for a write operation is to perform a small write. In this case, some blocks have

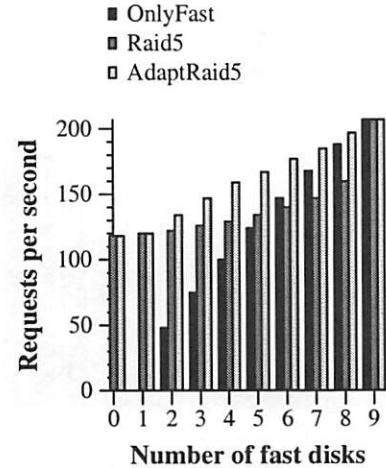


Figure 9: Writing 8Kbytes blocks (W8).

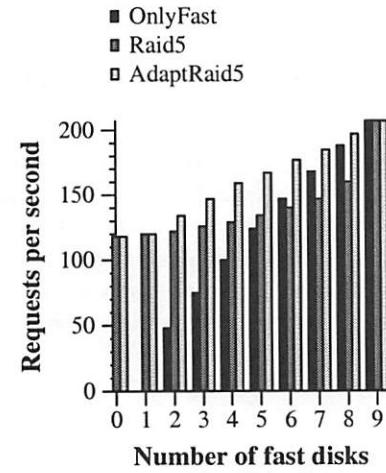


Figure 10: Writing 256Kbytes blocks (W256).

to be read in order to compute the parity of the stripe. This situation is different from the previous one, besides introducing the issue of the extra reads, because requests do not use all disks and this increases the parallelism between requests. This extra parallelism can be important in configurations with few fast disks because this parallelism will not be exploited by AdaptRaid5 and OnlyFast when only fasts disks are used, while it will be exploited by RAID5 that always uses all disks.

To do this evaluation we have measured the number of requests per second achieved by each evaluated system when 8Kbytes and 256Kbytes requests are done (workloads W8 and W256 described in Section 5.2) (Figures 9 and 10).

In this case, AdaptRaid5 is also better than RAID5, for the same reason as before. It knows how to use

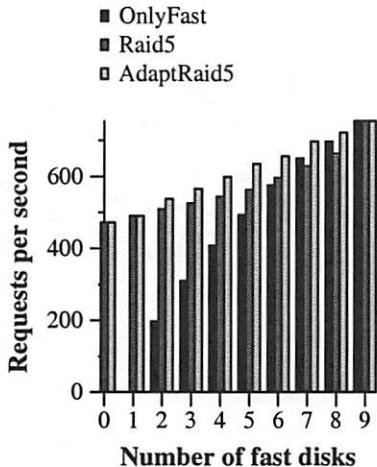


Figure 11: Reading 8Kbytes blocks (R8).

the fast disks. Furthermore, we can also see that the extra parallelism RAID5 can exploit is not enough compared to the benefit of only using fast disks for many of the requests.

When we compare AdaptRaid5 with OnlyFast, we observe that our proposal has a better performance than OnlyFast. This happens because AdaptRaid5 can use more disks and it can take advantage of the parallelism between requests.

6.4 Read Performance

Once the write performance has been evaluated, we need to measure the performance obtained by read operations. This evaluation has been done measuring the number of requests per second obtained by the system when requesting read operations 8Kbytes, and 2048Kbytes long (workloads R8 and R2048 described in Section 5.2). These results are presented in Figures 11 and 12.

In the first case (Figure 11), where requests are 8Kbytes, we observe a very similar behavior as in the previous cases. The only difference is that the performance of RAID5 and OnlyFast gets closer to AdaptRaid5 than in previous experiments. This happens because on these read operations, only one disk is used per request and more parallelism between requests can be achieved by OnlyFast and the probability of using a slow disk decreases in RAID5.

In the second case (Figure 12), the requests are much larger and this has two effects. If we observe

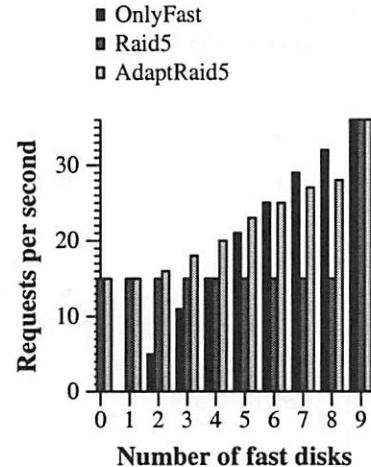


Figure 12: Reading 2048Kbytes blocks (R2048).

RAID5 performance, it remains unmodified when more fast disks are added. This is because all disks are used in the request and thus, slow disks are always included. If we focus on OnlyFast, we can see that it outperforms AdaptRaid5 when more than 6 fast disks are used. This happens because when these many fast disks are used, OnlyFast has enough parallelism within a request to obtain a good performance. On the other hand, AdaptRaid5 has to handle slow disks in many of the requests slowing down its performance. This means that if enough fast disks are used and only large reads are to be done, AdaptRaid5 is not the best solution.

6.5 Real-Workload Performance

The last experiment consists of running the trace file from HP described in Section 5.2. These results are presented in Figure 13. In this graph, we present the performance gains (in %) obtained by our distribution algorithm when compared to RAID5 and OnlyFast. The graph is divided in two parts. The left part shows the gain for read operations and the right part presents the results for write operations.

As expected, our algorithm is significantly faster than the other ones tested. The reasons are the same ones we have been discussing so far. The only exception is when 8 fast disks are used. In this case, OnlyFast is faster as it can achieve enough parallelism between requests and no slow disks are ever used. Nevertheless, maintaining only one slow disk does not seem to be very reasonable, and in this case we would recommend to discard the old disk

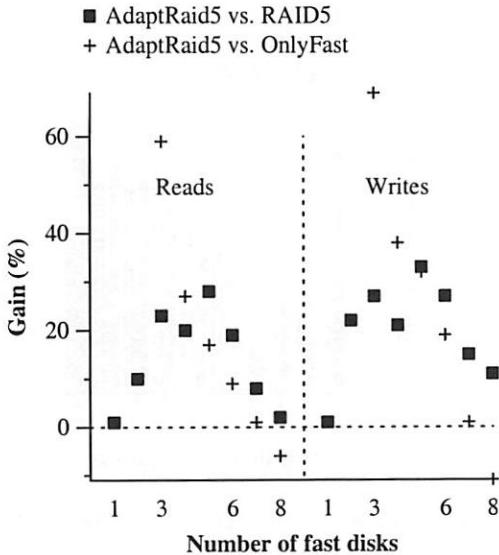


Figure 13: Performance gain of AdaptRaid5 over the rest of configurations in a real workload.

(unless the capacity is needed.)

6.6 Sensitivity Analysis of the UF Parameter

In all the experiments run so far, we have used UF values that maximize the utilization of the disks as far as capacity is concerned. Now, we want to see how sensitive is the performance of the array to the different values of UF. For this reason, we have tested the HP99 workload varying the UF factors on different array configurations. The different array configurations have 9 disks, but the number of fast disks used varies. These different configurations are represented by the different curves presented in Figures 14 and 15. The combinations of UF values used range, on the one hand, from $UF_{fast} = 1$ and $UF_{slow} = .1$ to both $UF_{fast} = UF_{slow} = 1$. These tests have been marked in Figures 14 and 15 using the name $S = .X$, which represents the value of UF_{slow} because UF_{fast} remains 1 all the time. On the other hand, we have also tried a couple of configurations where the slow disks have higher UF values. In these two tests, $UF_{slow} = 1$ and UF_{fast} takes .9 and .8 as values. These experiments are marked using the name $F = .X$, which represents the value of UF_{fast} because UF_{slow} remains 1 all the time

Figure 14 presents the average read times obtained in these experiments. The first thing we can observe

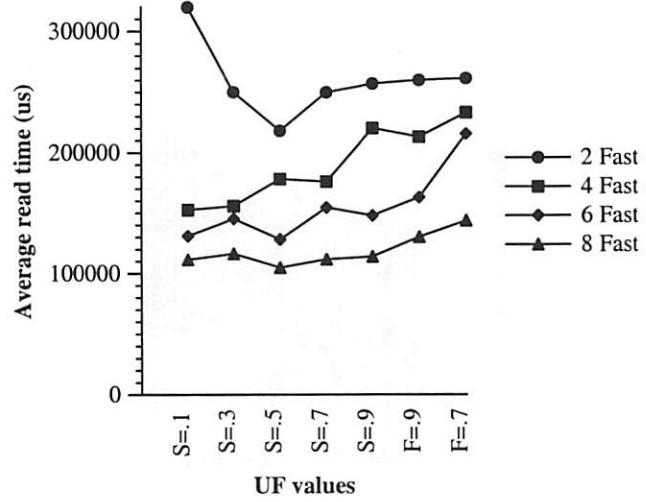


Figure 14: Variation in the average READ time when changing the UF factors for different disk configurations.

is that, in general, the more the slow disks are used, the longer it takes to perform read operations. The exception to this rule appears when only a few fast disks are used. In this case, the higher speed of fast disks cannot outweigh the parallelism obtained by the larger number of slow ones and the best read access time is achieved when slow disks are used half the time the fast ones.

It is also important to notice, that the curves are not perfect because there are other parameters that also have their effect in the performance. Changing the UF values also changes the placement of data and parity blocks, which also has an effect in performance.

Figure 15 presents the results of the same experiments, but for the average write time. In this figure we can observe the same behavior as with read operations.

Summarizing, the election of UF values is especially important if the number of fast disks is small and the higher performance of the fast disks cannot outweigh the parallelism of the large number of slow disks. Otherwise, using the fast disks as much as possible seems to be the way to go. Nevertheless, this election should also take capacity into account because different UF values achieve arrays with different capacities.

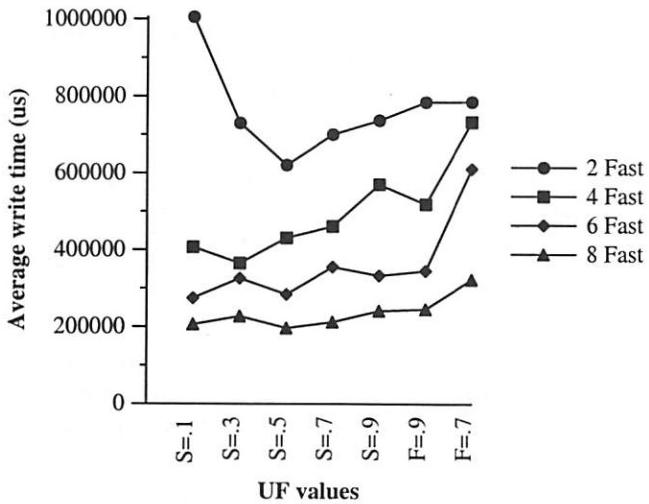


Figure 15: Variation in the average WRITE time when changing the UF factors for different disk configurations.

7 Future Work

In the future, we plan to concentrate our work on these issues:

- Find the best block size for this kind of algorithm as was done for regular disk arrays by Chen [2].
- Implement the algorithm in the Linux kernel.
- Study mechanisms to allow adding/replacing disks while the array is on-line.

8 Conclusions

In this paper, we have presented AdaptRaid5, a block-distribution policy that takes full advantage of heterogeneous disk arrays.

This algorithm achieves a significant performance compared to the policies currently being used. We have proven this by using both synthetic and real loads.

Furthermore, we have also shown that arrays using AdaptRaid5 are able to serve many more disk requests per second than when blocks are distributed assuming that all disks have the lowest common speed, which is the solution currently being used.

Finally, we have to keep in mind that this algorithm has been evaluated for an array built from disks attached to a SAN, but it would also work in other array configurations.

9 Availability

Other papers and reports about heterogeneous disk arrays

- people.ac.upc.es/toni/papers.html

Simulator information and downloading

- people.ac.upc.es/toni/software.html

A simplified version of the algorithm in pseudo code

- people.ac.upc.es/toni/AdaptRaid/pcAR5.html

Acknowledgments

We thank the Storage System Group at HP Laboratories (Palo Alto), and especially to John Wilkes, for letting us use their 1999 disk traces and for their interesting comments. We are also grateful to all the anonymous referees whose comments helped us to improve the quality of this paper. Finally, we thank Carla Ellis, who has been our shepherd and has shown us many ways to improve the quality of this work.

References

- [1] ANDERSON, T. E., DAHLIN, M. D., NEEFE, J. M., PATTERSON, D. A., ROSELLI, D. S., AND WANG, R. Y. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles* (December 1995), pp. 109–126.

- [2] CHEN, P., AND LEE, E. K. Striping in a RAID level 5 disk array. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May 1995), pp. 136–145.
- [3] CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. RAID: High-performance and reliable secondary storage. *ACM Computing Surveys* 26, 2 (1994), 145–185.
- [4] CORTES, T., AND LABARTA, J. HRaid: A flexible storage-system simulator. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications* (June 1999), CSREA Press, pp. 772–778.
- [5] CORTES, T., AND LABARTA, J. A case for heterogenous disk arrays. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster'2000)* (November 2000), pp. 319–325.
- [6] DAN, A., AND SITARAM, D. An online video placement policy based on bandwidth to space ratio (bsr). In *Proceedings of the SIGMOD* (1995), pp. 376–385.
- [7] GROCHOWSKI, E., AND HOYT, R. F. Future trends in hard disk drives. *IEEE Transactions on Magnetics* 32, 3 (May 1996).
- [8] HARTMAN, J., AND OUSTERHOUT, J. K. The zebra striped network file system. *Transactions on Computer System* 13, 3 (1995), 274–310.
- [9] HOLLAND, M., AND GIBSON, G. A. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the 5th Conference on Architectural Support for Programming Languages and Operating Systems* (October 1992), pp. 23–35.
- [10] HU, Y., AND YANG, Q. A new hierarchical disk architecture. *IEEE Micro* (November/December 1998), 64–75.
- [11] KOTZ, D., TOH, S. B., AND RADHAKRISHNAN, S. A detailed simulation model of the HP-97560 disk drive. Tech. Rep. PCS-TR94-220, Department of Computer Science, Dartmouth College, July 1994.
- [12] LEE, E. K., AND KATZ, R. H. The performance of parity placements in disk arrays. *IEEE Transactions on Computers* 42, 6 (June 1993), 651–664.
- [13] MCKUSICK, M., JOY, W., LEFFLER, S., AND FABRY, R. A fast file system for unix. *ACM Transactions on Computer Systems* 2, 3 (August 1984), 181–197.
- [14] MCVOY, L., AND KLEIMAN, S. Extent-like performance from a unix file system. In *Proceedings of the Summer Technical Conference* (June 1990), USENIX Association, pp. 137–144.
- [15] RUEMMLER, C., AND WILKES, J. Unix disk access patterns. In *Proceedings of the Winter USENIX Conference* (January 1993), pp. 405–420.
- [16] RUEMMLER, C., AND WILKES, J. An introduction to disk drive modeling. *IEEE COMPUTER* (March 1994), 17–28.
- [17] SANTOS, J. R., AND MUNTZ, R. Performance analysis of the rio multimedia storage system with heterogenous disk configurations. *ACM Multimedia* (1998), 303–308.
- [18] SEAGATE. Segate web page. <http://www.seagete.com>, January 2000.
- [19] SMITH, K. A., AND SELTZER, M. A comparison of ffs disk allocation policies. In *Proceedings of the Annual Technical Conference* (January 1996), USENIX Association.
- [20] STODOLSKY, D., GIBSON, G., AND HOLLAND, M. Parity logging overcoming the small write problem in redundant disk arrays. In *Proceedings of the 21th Annual International Symposium on Computer Architecture* (1993), pp. 64–75.
- [21] VEPSTAS, L. Software-raid howto. <http://www.linux.org/help/lpd/howto/Software-RAID-HOWTO.html>, 1998.
- [22] WILKES, J. Personal communication, September 1999.
- [23] WILKES, J., GOLDING, R., STAELIN, C., AND SULLIVAN, T. The HP AutoRAID hierarchical storage system. In *Proceedings of the 15th Operating System Review* (December 1995), ACM Press, pp. 96–108.
- [24] ZIMMERMANN, R. *Continuous media placement and scheduling in heterogeneous disk storage systems*. PhD thesis, University of Southern California, December 1998.

Reverse-Engineering Instruction Encodings

Wilson C. Hsieh

University of Utah

wilson@cs.utah.edu

Dawson R. Engler

Stanford University

engler@csl.stanford.edu

Godmar Back

University of Utah

gback@cs.utah.edu

Abstract

Binary tools such as disassemblers, just-in-time compilers, and executable code rewriters need to have an explicit representation of how machine instructions are encoded. Unfortunately, writing encodings for an entire instruction set by hand is both tedious and error-prone. We describe DERIVE, a tool that extracts bit-level instruction encoding information from assemblers. The user provides DERIVE with assembly-level information about various instructions. DERIVE automatically reverse-engineers the encodings for those instructions from an assembler by feeding it permutations of instructions and analyzing the resulting machine code. DERIVE solves the entire MIPS, SPARC, Alpha, and PowerPC instruction sets, and almost all of the ARM and x86 instruction sets. Its output consists of C declarations that can be used by binary tools. To demonstrate the utility of DERIVE, we have built a code emitter generator that takes DERIVE's output and produces C macros for code emission, which we have then used to rewrite a Java JIT backend.

1 Introduction

Binary tools such as assemblers, debuggers, disassemblers, dynamic code generation systems [3, 8, 10, 15, 18], just-in-time compilers [4, 7, 12], and executable code rewriters [14, 21, 24] need to contain a representation of how machine instructions are encoded. For example, a JIT needs to know that the x86 instruction `addl %ecx, %ebx` corresponds to the bits `0x01cb` (formed by a bitwise OR of the `addl` opcode `0x01c0` with `0x8` for the `%ecx` argument and `0x3` for the `%ebx` argument). Unfortunately, specifying instruction encodings with current tools requires looking up and detailing the offsets, sizes, and values of many instruction fields. Unsurprisingly, this process is both error-prone and tedious, especially for CISC machines such as the x86.

Currently, system builders must transcribe in-

A short version of this paper that only described the DERIVE solver appeared in the proceedings of the DYNAMO 2000 workshop [6].

struction encodings from an architecture reference manual. We have built a tool called DERIVE that eliminates the need to specify many of the bit-level details of instruction layout. The user supplies DERIVE with an assembly-level description of an instruction set: instruction names and operand types (registers, immediates, or labels). DERIVE outputs a description of how each instruction is encoded, which is given in the form of C structure declarations. The user does not specify binary-level details such as operand widths, operand offsets, opcode values, and register value encodings. As a result, the potential for misspecification by the user is less than that with other tools.

DERIVE is based on a simple observation: virtually all architectures for which a programmer needs binary encodings will already have programs (assemblers) that contain this information. Therefore, we should be able to extract the information from the assembler, which is what DERIVE does. At a high level, DERIVE does so by feeding permutations of each instruction to the system's assembler, and doing equation solving on the assembler's output to determine how the instruction is encoded (its opcode and its operand encodings). The DERIVE implementation solves the entire MIPS, SPARC, Alpha, and PowerPC instruction sets. It handles most of the ARM and x86 instruction sets: it does not yet handle some instructions, such as those with non-continuous fields.

DERIVE produces C data structure declarations that describe how an instruction set is encoded. It would not be difficult to produce declarations in other languages. These declarations can be used by tools that interpret or manipulate binaries. As an example, we have built a code emitter generator that takes the reverse-engineered declarations and generates C macros for fast code emission. To demonstrate the utility of these tools, we have rewritten Kaffe's [25] JIT compiler to use these macros. Using our automatically generated code emitters reduced the size of the Kaffe backend description by 40%.

On a side note, DERIVE can be viewed as an assembler tester. Because of how it reverse-engineers instructions, it can quickly find differences between

what an architecture manual says, and what an assembler actually implements.

In Section 2 we discuss related work: Collberg’s compiler-reverse-engineering system and the New Jersey Machine-Code Toolkit. Section 3 describes our assumptions about instruction set encodings, and explains how DERIVE works. Section 4 discusses the code emitter generator that we have built on top of DERIVE. Section 5 summarizes our conclusions and describes our future work. Appendix A gives a complete input specification of the MIPS instruction set for DERIVE.

2 Related Work

The most important precedent to DERIVE is Collberg’s work on reverse-engineering compilers [1]. Collberg’s system does a “reverse interpretation” to infer what assembly instructions should be used to implement a high-level language. It runs pieces of mutated assembly code to see how the semantics of the instructions changes. His work is aimed at building automatically retargeting compilers, where the descriptions of the semantics of assembly instructions are used to automatically retarget the BEG back-end generator [5].

DERIVE is complementary to Collberg’s work. DERIVE computes information that can be used to bypass the assembler, which is important for building JITs and binary manipulation tools. DERIVE addresses a simpler problem domain than Collberg’s system, since it reverse-engineers a syntax transformation and not a semantic transformation. On the other hand, our problem domain is much more tractable: it takes on the order of minutes to hours for DERIVE to reverse-engineer an instruction set’s encoding, whereas Collberg’s system takes days to reverse-engineer an instruction set’s semantics.

The work whose goals most closely match DERIVE is the New Jersey Machine-Code Toolkit (NJT) [20]. The NJT automatically generates routines to manipulate machine-code from user specifications written in a language called SLED. SLED specifications are exact descriptions of instruction layout, written at several levels of abstraction. At the lowest level of description, SLED “fields” are used to describe instruction bitfields. For example, the description of the SPARC instruction fields in SLED reads as follows [19]:

```
inst 0:31 op 30:31 disp30 0:29 rd 25:29
op2 22:24 imm22 0:21 a 29:29 cond 25:28
disp22 0:21 op3 19:24 rs1 14:18 i 13:13
asi 5:12 rs2 0:4 simm13 0:12 opf 5:13
```

DERIVE could be used to eliminate the field level

of specification in NJT specifications. Users of DERIVE need only worry about an interface that is at the level of assembly language (the level at which NJT’s “constructors” are written). Removing such extra levels of detail eliminates some potential sources of specification error.

The authors of NJT use the assembler to do testing of specified encodings [9], by choosing random values from within the space of encodings. We turn the process around and use the assembler to derive encodings. Because a user of NJT specifies the various classes of encodings, the NJT specification tester does not need to search as much of the encoding space as DERIVE does.

NJT supports several features that DERIVE does not, such as support for relocation. For targets such as JITs, which are our primary interest, relocation is not an issue. As another example, NJT can handle synthetic assembly instructions cleanly. The current implementation of DERIVE does not yet handle complex synthetic instructions well; however, we could easily layer a tool that described synthetic instructions on top of DERIVE.

3 Implementation

DERIVE takes a high-level description of an instruction set and an assembler, and generates C header files that describe the instruction set. Figure 1 illustrates the tool chain for DERIVE. *idc* is described in Section 3.1; the solver is described in Section 3.2; and the code emitter generator is described in Section 4.

DERIVE extracts the encoding of each instruction by sampling a small number of possible input sets. We assume each instruction takes a fixed number of *fields* as operands. We assume that fields have three types: registers, immediates, and labels/jump targets. Registers are a (usually small) set of textual names; immediates are integer values that can range over a large set of possibilities (e.g., on the x86, some instruction take several 32-bit immediates); and labels are symbolic instruction addresses. These assumptions are sufficient to describe the instruction sets for which we have written specifications (which cover a wide range of modern ISAs).

We make the following assumptions about instruction encodings:

1. Assembly encodings are *stateless* transformations. That is, a particular instruction always has the same encoding. In addition, assembling two correct instructions together gives the same results as concatenating the results of as-

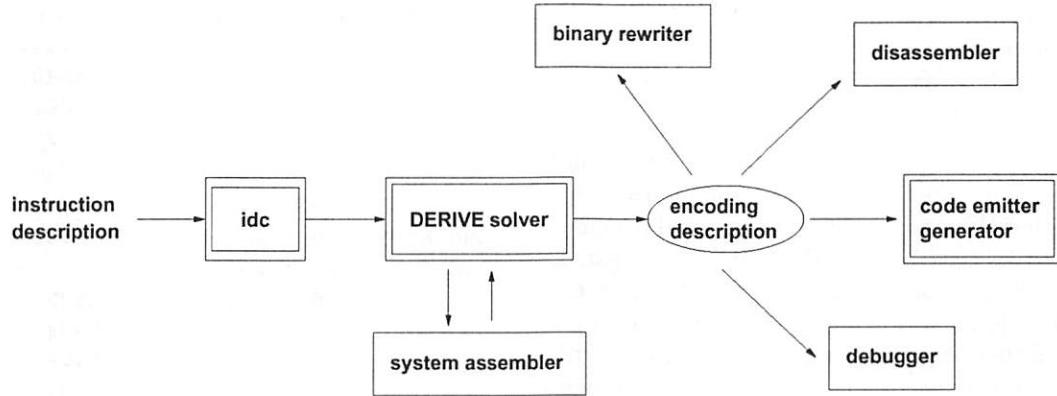


Figure 1: An example tool chain that uses DERIVE. We have built the tools that are in double boxes.

sembling them separately. To satisfy this assumption, assemblers must be prevented from adding, removing, or reordering instructions; in general, the user must provide DERIVE with the appropriate assembly directives. This assumption lets us batch together the assembly of many instructions into one file, which reduces solving time by a large constant. For example, on our machine (a dual-processor 600 MHz Pentium III), we can assemble 2000 instructions in just about twice as much time as we can assemble one instruction—process creation and file I/O dominate the cost of assembly.

2. We assume that the validity and encoding of a value in a field does not depend on the value of any other field. We can thus compute a field’s encoding by enumerating every legal value for it while holding all other fields constant. This assumption allows us to solve for each field separately. Some instructions in certain instruction sets (such as the ARM) do not satisfy this assumption, in which case DERIVE must do a more expensive search.
3. The field for an n -bit immediate whose high bit is set is independent of the low bits of the immediate. That is, constants between 2^{n-1} and $2^n - 1$ can all be encoded in the same instruction field. This assumption lets us solve for immediate fields more quickly; otherwise, our solver would have to enumerate all legal operand values.
4. Immediates are encoded with “simple” transformations. “Simple” means that an immediate’s value can have leading or trailing zeros removed, constants subtracted from it, or leading

one bits added (for sign extension). If DERIVE detects a complex transformation that it cannot represent (such as the scale factor in x86 memory instructions, which is encoded as the logarithm of the scale factor), it immediately stops and tells the user, who can provide DERIVE with the transformation explicitly. Our implementation currently makes the assumption that negative immediates are encoded using twos-complement; this assumption could be changed fairly easily.

3.1 Specifying Instructions

The front end to DERIVE, `idc`, translates assembly-level instruction set descriptions into the intermediate form that DERIVE uses. `idc` is about 900 lines of `lex` and `yacc` code. Users describe instructions with a `yacc`-like description that contains a list of instructions to generate and a format description that describes the assembly syntax and the operand types. Note that the formats are grouped by assembly syntax, not by encoding class; we believe that the former is far more readable, as it expresses logical relationships rather than encoding relationships.

The description of SPARC ALU instructions reads as follows. Note that the field-level information necessary with NJT does not have to be specified by the user.

```

iregs = ( %g0, %g1, %g2, %g3, %g4, %g5, %g6,
          %g7, %o0, %o1, %o2, %o3, %o4, %o5, %o6, %o7,
          %l0, %l1, %l2, %l3, %l4, %l5, %l6, %l7, %i0,
          %i1, %i2, %i3, %i4, %i5, %i6, %i7 );

```

```

and, andcc, andn, andncc, or, orcc, orn, orncc,
xor, xorcc, xnor, xnorcc, sll, srl, sra, add,
addcc, addx, addxcc, taddcc, taddcctv, sub,
subcc, subx, subxcc, tsubcc, tsubcctv, mulsc,

```

```

umul, smul, umulcc, smulcc, udiv, sdiv, udivcc,
sdivcc, save, restore
--> &op& r_1:iregs, r_2:iregs, r_dest:iregs
| &op& r_1:iregs, imm, r_dest:iregs ;

```

The production describes the assembler syntax and the types of the operands for multiple instructions. It states that there are register-register and register-immediate forms of all of the instructions listed. The placeholder `&op&` indicates where the instruction name appears. The operand specification says that the formatting substring ‘‘`r_1:iregs`’’ is replaced by members of the list of registers `iregs`. Several conventions apply: register field names begin with `r`, a jump target has the unique name `&label&`, and an immediate field is any other field.

3.2 The Solvers

DERIVE is composed of three solvers, each specialized to derive a specific operand type: the register solver solves for register fields/operands, the immediate solver solves for immediate fields, and the jump solver solves for jump target fields. Each solver uses the assembler to compute instruction encodings. DERIVE emits code into an assembler file, runs the assembler, and finds the code in the resulting executable. DERIVE tests for any difference in endianness between the target and the solving architectures, and appropriately swaps bytes in the object code before doing any solving.

To demarcate the machine code in the executable, DERIVE explicitly emits fenceposts around the code using assembler data directives. Our fenceposts are a randomly chosen sequence of bytes. It is unlikely that DERIVE emits a sequence of instructions that match the fencepost, since it does not exhaustively search the set of instructions: we only search the entire space of values for register fields, and not for immediate and jump fields. Although we do not currently do so, a simple way to detect and avoid fencepost conflicts would be to solve each instruction twice with different fencepost values.

DERIVE represents instruction encodings as an opcode mask and an arbitrary number of operand fields. The opcode mask contains the bitmask of 1’s that must be set in an instruction to specify a given opcode. Register fields are specified as a sequence of masks, one for each legal register value; immediates and label fields are represented as a size, an offset, and a simple transform.

The solvers all work in the same basic manner. Each one locates a field and determines its size by emitting one instruction for each legal operand value of that field (while holding all other fields’ values

and %g7 , %g6 , %g0;	0x8009 0xc006
and %g7 , %g6 , %g1;	0x8209 0xc006
and %g7 , %g6 , %g2;	0x8409 0xc006
and %g7 , %g6 , %g3;	0x8609 0xc006
and %g7 , %g6 , %g4;	0x8809 0xc006
and %g7 , %g6 , %g5;	0x8a09 0xc006
and %g7 , %g6 , %g6;	0x8c09 0xc006
and %g7 , %g6 , %g7;	0x8e09 0xc006
and %g7 , %g6 , %o0;	0x9009 0xc006
and %g7 , %g6 , %o1;	0x9209 0xc006
and %g7 , %g6 , %o2;	0x9409 0xc006
and %g7 , %g6 , %o3;	0x9609 0xc006
and %g7 , %g6 , %o4;	0x9809 0xc006
and %g7 , %g6 , %o5;	0x9a09 0xc006
and %g7 , %g6 , %o6;	0x9c09 0xc006
and %g7 , %g6 , %o7;	0x9e09 0xc006
and %g7 , %g6 , %l0;	0xa009 0xc006
...	

Figure 2: The assembly that DERIVE generates to solve for the last register field of the SPARC add instruction, and the resulting binary instructions that it analyzes.

fixed) and finding all bits that change in the binary encoding. Figure 2 illustrates this process for the last operand of the SPARC add instruction. Those bits that change belong to the field. Its offset is given by the lowest changing bit; its size by the difference between its lowest and highest bit. A specific operand’s value exactly equals the value of these bits when it is used. All other bits belong to other fields, or to the opcode mask.

As each solver is run, the opcode mask is refined. That is, each solver ‘‘claims’’ bits for various fields. After all fields have been solved for, the opcode mask is set to the remaining unclaimed bits (i.e., those that are set to 1 in every emitted instruction).

We make one general assembler-dependent assumption, which is that the assembler will produce errors ‘‘when expected.’’ For example, we assume that assemblers will return an error when illegal registers are used in an instruction, or when constants are too large. DERIVE finds the sizes of immediate fields by testing larger and larger immediates, and it expects that the assembler will eventually return an error message.

Unfortunately, assemblers do not always report errors when they should. `Gas` shows some unexpected behavior, in that it will accept some positive constants that are too large for signed fields (for example, for a 16-bit signed field it accepts constants between 32768 and 65535). In general, we try to re-

move such dependencies by checking the generated code: DERIVE gets around this particular bug by making sure that each constant value examined actually results in a different instruction.

3.2.1 Register Solver

The register solver is DERIVE's most basic solver, and is called by the other solvers. It has two tasks. First, it computes an instruction's opcode mask with respect to the register fields. The opcode mask consists of all of the bits that are not used by the register fields. Second, it finds both the location and size of each "register" operand field, along with the bitmask that must be set to specify a given operand value. A register operand is any operand for which the client enumerates the possible textual values.

The register solver works in the following manner for a particular register field:

1. Iterate over all legal registers in the field. For each register, create a copy of the instruction format string. Replace the operand with the register value, and all other operands with some legal values. Emit the instructions and read them back into a buffer `inst`.
2. While iterating over each register value, incrementally reduce the opcode mask by ANDing it with each binary instruction:

```
op_mask = ~0;
foreach r in registers
    op_mask &= inst[r];
```

This process will leave the mask with 1's in every bit that has a 1 set for all instruction instances.

3. Examine the emitted instruction stream, and look for all bits that change between 0 and 1. Such bits belong to the current field, since all other operand values were fixed, and all values for the field were enumerated.

To find these bits, bitwise AND each instruction with the complement of the opcode mask and logically summing the result:

```
field = 0;
foreach r in registers
    field |= (inst[r] & ~op_mask);
```

At the end, the `field` mask has a 1 set for every bit in the field. The size of the field is bounded by the most and least significant bits set in this computed mask. The field's offset is given by its least significant bit.

4. To derive the 1's that must be set to encode each register for the current field, iterate again over all legal registers, and bitwise AND each instruction with the computed field mask:

```
foreach r in registers
    fmask[r] = inst[r] & field;
```

We have implemented two main extensions to this simple scheme. First, on some architectures, a specific register value can change the actual instruction encoding. For example, on the x86, different instruction forms are used when the `%eax` register is used as an operand. The solver detects such discontinuities by checking that all instances of an instruction are of the same length. That is, when it emits the sequence of instructions (as shown in Figure 2), it checks that the number of instruction bytes emitted equals the number of instructions multiplied by the size of one instruction. If it does not, it solves for the register values independently, and emits multiple instruction specifications: each specification identifies which specific register values it corresponds to.

Second, the solver allows users to supply register operand lists that contain illegal values, which it automatically culls. This syntax is useful for situations where instructions only accept subsets of possible register values. For example, the SPARC architecture supports floating-point instructions that take different combinations of registers, depending on whether the inputs and outputs are single-, double-, or quad-precision. Specifying exactly which operands are legal would increase the size of the SPARC specification by about 40%, and would also increase the probability of error. Instead, clients can state that every floating-point instruction takes any floating-point register as an operand:

```
fregs = ( %f0, %f1, %f2, %f3, %f4, %f5, %f6,
    %f7, %f8, %f9, %f10, %f11, %f12, %f13, %f14,
    %f15, %f16, %f17, %f18, %f19, %f20, %f21,
    %f22, %f23, %f24, %f25, %f26, %f27, %f28,
    %f29, %f30, %f31 );

fadds, fsubs, fmuls, fdivs, faddd, fsubd,
fmuld, fdivd, faddq, fsubq, fmulq, fdivq,
fsmuld, fdmulq
    --> %op& r_1:fregs, r_2:fregs, r_3:fregs;
```

DERIVE automatically eliminates "bad" registers by first randomly selecting operand values until it finds a sequence that the assembler accepts. It then finds all legal values for a field by trying all of its values while holding the others operands fixed to legal values.

3.2.2 Immediate Solver

The immediate solver computes the width and position of each immediate field, and also any transformation of the values in immediate fields. The immediate solver is called after the jump solver, if there are relative jump targets; it is used directly to solve for absolute jump targets (since those are just transformed immediate values). We explain it before the jump solver, because its behavior is closer to that of the register solver.

The main difference between this solver and the register solver is that it is impractical to enumerate all legal values for an immediate operand: an n -bit immediate field would require 2^n permutations. Register fields tend to be small (on the order of 5 bits for modern architectures), whereas immediate fields can be substantially larger. As a result, we solve for each bit size of the immediate field, rather than each possible value.

The solver first finds an immediate field's size by iterating upwards from 1 bit, 2 bits, etc., until the assembler refuses to assemble the instruction. It then iterates down from the maximum number of bits (call it n) and solves for each immediate size.

The immediate solver works in the following manner for each bit size m :

1. Choose a random $m - 1$ bit value and create two m -bit constants, v_0 and v_1 , by setting v_0 to the value and v_1 to its complement. Then set the m th bit in both v_0 and v_1 :

```
x = (1 << (m - 1));
# randomize low m-1 bits
v0 = random() % x;
# v1 complements those bits
v1 = ~v0 % x;
# set mth bit in v0 and v1
v0 = v0 | x;
v1 = v1 | x;
```

By emitting instructions with these two constants, we force the low $m - 1$ bits of the immediate fields to have complementary bit values, while all other bits remain constant. Next, create two instructions, the first with the immediate operand replaced with v_0 , the second with v_1 . Use the register solver to derive the register field encodings for these two similar-looking instructions.

```
# copy instruction
inst0 = inst1 = inst;
# replace immediate operand with v0
rewrite(inst0.fmt, op.field_name, v0);
# replace immediate operand with v1
```

```
rewrite(inst1.fmt, op.field_name, v1);
# solve each copy
register_solve(s0, inst0);
register_solve(s1, inst1);
```

The two specifications s_0 and s_1 should have the same register masks, and the same size in bytes. The only difference between the two should be the opcode masks computed by the register solver, which will differ by exactly the bits that differ in v_0 and v_1 .

2. Find the immediate field by first XORing the two opcode masks. Since only the lower $m - 1$ bits of the field differ, this action sets 1's exactly in the location of these bits and 0s everywhere else; we add the m th bit in explicitly (note that we assume that this bit is contiguous with the rest of the field). The least significant bit gives the field's offset. Refine the opcode mask by removing all field bits from it.

```
# set the low m-1 bits in field
field_bits = s0.op_mask ^ s1.op_mask;
# find least significant bit
field_offset = lsb(field_bits);
# add back mth field bit
field_bits |= 1 << (m + field_offset - 1);
# fix opcode mask
s0.op_mask = s0.op_mask & ~field_bits;
```

3. Check the value of the field against the value of the encoded field to see if any simple transformations are used, such as a shift to the right by a small constant.
4. Check to see if a previous encoding matches this one. Since we work our way down from larger-valued immediates to smaller-valued immediates, we may have already discovered the encoding for the field. For example, a SPARC 13-bit signed immediate field encodes all signed values between 1 bit and 13 bits. In contrast, on the x86 4-byte memory displacements are encoded differently than 2-byte memory displacements.

If we have already found an encoding, the current encoding is ignored; otherwise, it is added to the list of derived encodings. We must evaluate the encoding for each immediate size, because some instruction encodings on certain architectures (x86) vary with the size of the immediate provided.

Our actual solver is more general than this sketch, and handles instructions with an arbitrary number of immediate operands. For instructions with more

than one immediate operand, any variable-length immediate fields are not independent of each other—the length of one will affect the position of any others. Therefore, DERIVE cannot solve for immediate fields separately.

The current implementation has two limitations. First, it assumes that every immediate value that fits in a given field is legal: that is, there are no “holes” in the value space for an immediate field. This restriction is not a real problem. Second, it does not handle immediates that are encoded as multiple non-contiguous bit ranges. It should not be difficult to extend DERIVE to handle such immediates.

3.2.3 Jump Solver

The jump solver derives the encoding for relative jump target fields (labels). Jumps can be classified in two ways: (1) relative vs. absolute jumps and (2) jumps that take immediates vs. those that only take labels. The jump solver finds encodings for relative jumps that take labels as operands. Jumps that accept immediate operands are handled by invoking the immediate solver. Absolute jumps that only take labels do not seem to occur in practice.

The difference between the jump solver and the immediate solver is that the jump solver must generates values (labels) differently. To set all bits in an n -bit immediate field, the immediate solver can emit the immediate directly; the jump solver may have to place a label about n instructions away. As a result, it would be impractical to solve for large offsets directly. We assume that backwards jumps are encoded using twos-complement, so that we can solve for the sizes of offsets.

DERIVE computes whether a jump is absolute or relative by emitting two consecutive jumps to the same target and comparing the emitted code values. Absolute jumps will have identical bits, since both instructions encode the same target address. Relative jumps will differ, since they are different distances from the target and thus will have different offset values.

Given an instruction that is a relative jump, the solver must find the jump target field’s width and offset in the instruction, starting point (how many bytes a jump of 0 bytes actually jumps from the jump instruction), and minimum jump size. The target address of the jump is $\text{target} = \text{start} + \text{encoded field} \times \text{jump size}$.

1. Find the minimum jump size by emitting jumps of 1, 2, ... bytes until the jump field changes.
2. Find the starting point for the jump (the place-

ment of the label that results in an offset of 0). The starting point is found by searching for a label placement that results in an instruction i , where ORing against i is the identity function. In other words, emit jumps to different labels around the jump, and OR each of the jumps against the others to find one whose offset field must be all 0’s.

3. Find the label’s offset in the instruction by emitting a forward jump just past the starting point for the jump. These two instructions will differ by a single bit, which is the lowest bit in the target field.
4. Find the label’s size in the instruction by emitting a negative-offset label. Assuming that relative jumps are encoded using twos-complement (or even ones-complement), this label sets the high bit of the label. We assume that the label is contiguous in the instruction, and that it consists of the bits between the label’s high and low bits.
5. Finally, determine how the jump distance is encoded by referencing labels at known offsets and comparing the field’s value to these locations. We check for the following transformations: subtraction by a constant, truncation of trailing zeros, or having leading bits truncated. For example, truncation of trailing zeros from a byte offset happens in SPARC jump instructions, since instructions are word-aligned.

Like the other solvers, the jump solver assumes that label fields are contiguous. It would be challenging to deal with non-contiguous label fields, because of the need to generate labels at large distances from jumps.

3.3 User Extensions

Some instruction sets (such as the ARM) do not satisfy all of the assumptions we have described. In addition, incorrect assemblers can provide bad information to DERIVE. We give the user mechanisms to address these problems: the user can inform DERIVE of complex immediate encodings, tell it when register field values may depend on each other, and provide explicit field widths when necessary. Table 1 lists the cases where we have needed to use these mechanisms.

The following description fragment shows how a user can specify complex encodings. The user provides C code that translates between the value that is mapped into the immediate field and the input

Violated Assumption	Architecture	Instruction Class
Fields values are independent	ARM	Register pre-/post-indexed addressing modes Base and index registers must differ
	PowerPC	Load multiple instructions Address register must not be a target
	PowerPC	Update instructions Base register must not equal target register
Simple transforms	x86	Scale factor in memory addressing Scale is encoded as a log
	SPARC	Sethi argument must have 10 low zero bits
	ARM	Certain addressing modes expect offset $\times 4$

Table 1: Exceptions that we have found to our encoding model. DERIVE’s hooks let clients easily extend its model to handle these cases. For fields that depend on each other, users annotate dependent registers in their specification, and supply a function that takes a list of symbolic register names and returns TRUE iff they are a legal combination. Users add missing transformations by providing a function that takes an immediate and returns the transformed value, and annotating immediates that use this encoding in the specification.

that the assembler expects. In this example, the x86 instruction encodes the logarithm of the scale factor that is given to the assembler. (This example could also be handled by making the scale factor a register type and enumerating the possible scale values.)

```
%{
unsigned pow2(unsigned x) { return 1 << x; }

ops_2_mem --> &op& r_1:regs,
disp(base:regs, index:index_regs, scale:pow2);
```

The following example shows how a user provides information about non-independent fields. For the PowerPC instructions mentioned, the base register *ra* must not be the same as the target register *rd*.

```
%{
bool update_disp(char *args[])
{
    char *rd = args[0], *ra = args[1];
    if (!strcmp(ra, "r0") || !strcmp(ra, rd))
        return FALSE;
    else return TRUE;
}

lbzu, lhzu, lhau, lwzu -->
&op&:update_disp R_d:regs, disp(R_a:regs);
```

Finally, DERIVE provides hooks that enable the user to overcome some assembler bugs that we have encountered. For example, GNU as allows the SPARC V9 ticc instruction to take an immediate that is too large. DERIVE reports that fields overlap, and the user can explicitly tell DERIVE the field width of 7 bits as follows:

Processor	Run time (minutes)	Description length (lines)
Alpha	6.3	104
ARM	$\approx 43.$	227
MIPS	2.5	81
PowerPC	4.8	186
SPARC	4.8	97
x86	$\approx 240.$	221
x86-kaffe	4.9	106

Table 2: The time it takes derive to run through each architecture, and how long our architecture descriptions are. x86-kaffe is the subset of x86 we needed to retarget Kaffe’s JIT to use DERIVE-generated emitters.

```
tgu, tleu, tcc, tcs, tpos, tneg, tvc, tvs
--> &op& r_c:cc, imm::7;
```

3.4 Using DERIVE

Table 2 summarizes the times it takes for DERIVE to run on several instruction sets, and also shows the length of DERIVE’s specifications. As the number of architectures in the table shows, DERIVE’s assumptions survive well under use. The assumptions in our model make DERIVE reasonably fast. ARM is slow because some of its instruction addressing modes violate the independence assumption. As a result, solving those instructions takes an inordinate amount of time, because DERIVE must check all combinations of register values. x86 is slow because of the large number of instructions and addressing

modes, as well as the special encodings for certain registers. The subset of the x86 ISA necessary to retarget the Kaffe JVM’s [25] JIT was small enough to run quickly.

While using DERIVE to reverse-engineer several instruction sets, we have come across several errors or inconsistencies in `gas` and various architectural manuals. The following list describes these errors and inconsistencies, and demonstrates that DERIVE’s reverse-engineering methodology can also be viewed as a useful testing methodology.

- GNU `as` does not assemble the Alpha `wh64` instruction.
- GNU `as` does not handle the Alpha rounding/trapping modes of floating-point `sqrt` instructions.
- GNU `as` on MIPS silently truncates the top bits of absolute addresses larger than 28 bits.
- GNU `as` does not quite handle setting the user mode bit in ARM addressing mode 4 correctly.
- GNU `as` accepts immediates that are too large for SPARC `ticc` instructions.
- GNU `as` often accepts n -bit positive values for n -bit signed immediate operands.
- “See MIPS Run” [22], Table 8.6, is incorrect for `mtc1` and `dtc1`.
- In the ARM manual [13], addressing mode 3 of register pre-/post-indexed instructions do not have the same listed restrictions as those same instructions in addressing mode 2, although `gas` enforces those restrictions.
- The Alpha manual [2] description of the `cvtst` (IEEE conversion) instruction seems incorrect, because it lists the `/s` suffix (a VAX rounding mode) as an option.

4 Using DERIVE’s Output

An important motivation behind building DERIVE was our desire to avoid hand-specifying the x86 instruction set. This distaste was an important reason why we have not retargeted two of our JIT systems [7, 8] to the x86, despite repeated requests.

One use of DERIVE-generated specifications is to generate code emitters from them. We have written an emitter generator that processes instruction specifications and generates C procedures or macros

that can emit instructions into a code buffer. Figure 3 shows the structure declaration for DERIVE’s output. Figure 4 shows a sample specification that DERIVE generates for the MIPS `break` instruction, which causes a breakpoint. Each instruction specification is transformed into a function or macro whose arguments are an index into the code buffer and the registers, immediates, or labels that are operands for that instruction. The choice of macros allows the compiler to propagate constants if parameters, such as register values, are known at compile-time.

The following example shows the macro generated for the x86 `addl` instruction, and how it is used:

```
#define E_addl_rr_1(_code, rf, rt) do {\\
    register unsigned short _0 = (0xc001\\
        | (((rf)) << 11))\\
        | (((rt)) << 8));\\
    *(unsigned short*)((char *)_code) = _0;\\
    _code = (void *)((char *)_code + 2);\\
} while(0)

/* emit "addl %ecx, %ebx" in code_buffer */
E_addl_rr_1(code_buffer, REGecx, REGebx);
```

With simple heuristics, we are able to generate code automatically that in most cases is as efficient and readable as the code a human would write. There are two challenges in generating efficient emitter code. First, we try to keep the number of arithmetic and bit shift operations on the instruction’s operands small. Second, we try to minimize the number of stores to the code buffer. For architectures with constant instruction lengths, such as all RISC architectures, our generated emitters incur one memory write per instruction generated.

Operands to an instruction are not always known when the instruction is generated. Dynamic code generators, for example, do not know the value of a forward-referenced label. Such a label must be patched later by a simple linker, once the actual value of the operand becomes known. DERIVE allows instruction operands to be marked so that they are not used in the emitter macro. Instead, the emitter generator generates an additional macro that can be called to fill in the operand. DERIVE clients can build their own linkers on top of this mechanism.

Our emitter generator can emit extra debugging information. First, our emitter generator is able to generate emitters that check the validity of their arguments. This feature is useful to catch bugs as early as possible during dynamic code generation. In addition, the generator can also produce emitters that print a textual description of each instruction generated during dynamic code generation.

```

struct inst_spec {
    char *inst;                                /* instruction name */
    char *fmt;                                 /* assembly format */
    unsigned short n_ops;                      /* operand count */
    unsigned nbytes;                           /* instruction size */
    char *namesuffix;                         /* suffix used by emitter generator */
    unsigned char op_mask[MAX_BYTES];          /* opcode mask */
};

struct operand {
    char *sym_op;                            /* symbolic name of operand */
    enum op_type { REG, IMMED, LBL_R, LBL_A } type;      /* operand type */
    t_type encoding;                         /* is immediate transformed? */
    unsigned lo;                             /* lowest legal value of field */
    unsigned nbits;                          /* number of bits in field */
    unsigned mask[MAX_REGS];                /* field mask */
    int offset;                            /* offset in instruction */
    sign_type signed_field;                /* signed or unsigned field? */
    int relative_offset;                   /* where relative jumps start
                                                from: offset from end of jump */
    int wants_ref;                         /* generate a separate emitter
                                                to set this field */
} ops[MAX_OPS];
};

```

Figure 3: The C encoding description that DERIVE outputs. It includes the instruction, formatting string used to generate the instruction, the instruction mask, and a list of operand specifications. *t_type* is an *enum* that represents some simple transformations on immediates. The output of DERIVE could be modified for use with other languages.

```

{
    "break",
    "&op& imm",
    1,
    4,
    "",
    { 0xd, 0x0, 0x0, 0x0, },
    { { "imm",
        IMMED,
        IDENT,
        0,
        10,
        { 0 },
        16,
        I_UNSIGNED,
        0,
        0
    },
}
/* suffix used by emitter generator */
/* opcode mask */
/* name of operand */
/* type of operand */
/* operand transformation */
/* lowest legal value */
/* number of bits */
/* field mask */
/* offset in instruction */
/* unsigned */
/* ignored for non-jumps */
/* do not generate an extra emitter */

```

Figure 4: DERIVE-generated specification for the MIPS break instruction.

The emitters are also useful as a means of testing DERIVE. For each instruction, we generate the emitter and a test program that invokes it with a given subset of parameters. The instruction encodings generated by the emitter function are then compared to the output generated by the target assembler. This procedure allows us to test DERIVE without actually running code on the target platform. DERIVE's emitter generator can cross-compile between architectures whose endianesses match.

To demonstrate that our tools work, we have retargeted Kaffe's [25] x86 JIT backend to use automatically generated emitters from just a subset of the x86 ISA. We reduced the number of lines in the backend description from 2,084 to 1,267. We also discovered that the original coders missed shorter instruction encodings in one case. Retargeting the JIT took approximately one day, which indicates that the emitter functions generated by DERIVE are usable in real applications.

DERIVE-generated emitters can be used in several other ways to support dynamic code generation. For example, they can be used to support back end construction for general-purpose dynamic code generation systems such as vcode [8] and ccg [17]. These systems provide assembly-like interfaces to C for dynamic code generation, and need encoding information to actually generate code. DERIVE could also be used to compute templates for application-specific systems such as DPF [7]. Templates can be specified in terms of symbolic instruction sequences, fed to DERIVE to get the corresponding binary encoders, and then reincorporated into the system.

5 Conclusions

The DERIVE system reverse-engineers instruction encodings from the system assembler. Users need only give assembly-level information about the instructions for which they want encodings, and not low-level information about bitfield layout. DERIVE successfully reverse-engineers instruction encodings on the SPARC, MIPS, Alpha, ARM, PowerPC, and x86. In the last case, it handles variable-sized instructions, large instructions (16 bytes), multiple instruction encodings determined by operand size, and other CISC features. As a proof of its utility, we have built a code emitter generator on top of DERIVE.

We plan to extend DERIVE's techniques to reverse-engineer object code file formats, including debugging and linkage information. Such information will enable us to build a set of reverse-engineered tools, including versions of ATOM [21], dynamic linking libraries [11], object-level sandboxes [23], executable

optimizers, and linkers. Builders of such tools are plagued by the need to repeatedly reimplement functionality contained in existing software. For some systems, it is too expensive to call existing programs: dynamic code generation systems cannot afford the time to call an assembler. In other cases, the software has an inappropriate form and must be rewritten from scratch. For example, one common "trick" that commercial companies use to discourage third-party vendors is to have proprietary symbol table layouts, which change on every software release [16]. The cost of manually reverse-engineering these formats has forced some implementors to avoid object-level modifications, in spite of the strong advantages for such an approach.

The source code for the current version of DERIVE is freely available at the following URL: <http://www.cs.utah.edu/~wilson/derive.tar.gz>.

Acknowledgments

We thank John Carter, David Chen, Eddie Kohler, Max Poletto, Patrick Tullmann, various anonymous referees, and our shepherd Vern Paxson for their comments on earlier drafts of this paper. Wilson Hsieh was supported in part by an NSF CAREER award, CCR-9876117. Dawson Engler was supported in part by DARPA contract MDA904-98-C-A933 and by a Terman Fellowship. Godmar Back was supported by DARPA contract F30602-99-1-0503.

References

- [1] C. Collberg. Reverse interpretation + mutation analysis = automatic retargeting. In *Proc. of PLDI '97*, June 1997.
- [2] A. A. Committee. *Alpha Architecture Reference Manual*. Digital Press, third edition, 1998.
- [3] C. Consel and F. Noel. A general approach for run-time specialization and its application to C. In *Proc. of 23rd POPL*, St. Petersburg, FL, Jan. 1996.
- [4] P. Deutsch and A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proc. of 11th POPL*, pp. 297–302, Salt Lake City, UT, Jan. 1984.
- [5] H. Emmelmann, F.-W. Schröer, and R. Landwehr. BEG—a generator for efficient back ends. In *Proc. of PLDI '89*, pp. 227–237, 1989.

- [6] D. Engler and W. Hsieh. DERIVE: A tool that automatically reverse-engineers instruction encodings. In *Proc. of DYNAMO*, Boston, MA, Jan. 2000.
- [7] D. Engler and M. Kaashoek. DPF: Fast, flexible message demultiplexing using dynamic code generation. In *Proc. of SIGCOMM 1996*, pp. 53–59, Stanford, CA, USA, Aug. 1996.
- [8] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proc. of PLDI '96*, pp. 160–170, Philadelphia, PA, USA, May 1996.
- [9] M. Fernández and N. Ramsey. Automatic checking of instruction specifications. In *Proc. of ICSE*, 1997.
- [10] B. Grant, M. Philipose, M. Mock, C. Chambers, and S. Eggers. An evaluation of staged run-time optimizations in DyC. In *Proc. of PLDI '99*, pp. 293–304, Atlanta, GA, May 1999.
- [11] W. W. Ho and R. A. Olsson. An approach to genuine dynamic linking. *Software: Practice and Experience*, 24(4):375–390, Apr. 1991.
- [12] U. Hözle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proc. of PLDI '94*, pp. 326–335, Orlando, Florida, June 1994.
- [13] D. Jaggar, editor. *ARM Architecture Reference Manual*. Prentice Hall, 1996.
- [14] J. R. Larus and T. Ball. Rewriting executable files to measure program behavior. *SPE*, 24(2):197–218, feb 1994.
- [15] M. Leone and P. Lee. Optimizing ML with run-time code generation. In *Proc. of PLDI '96*, May 1996.
- [16] S. Lucco. Personal communication. Use of undocumented proprietary formats as a technique to impede third-party additions, Aug. 1997.
- [17] I. Piumarta. ccg: dynamic code generation for C and C++. <http://www-sor.inria.fr/projects/vvm/ccg/>, Mar. 1999.
- [18] M. Poletto, W. Hsieh, D. Engler, and M. F. Kaashoek. ‘C and tcc: A language and compiler for dynamic code generation. *TOPLAS*, 21(2):324–369, Mar. 1999.
- [19] N. Ramsey and M. Fernández. New Jersey machine-code toolkit architecture specifications. <http://www.eecs.harvard.edu/nr/toolkit/specs/specs.ps>, Nov. 1996.
- [20] N. Ramsey and M. F. Fernández. The New Jersey machine-code toolkit. In *1995 Winter USENIX*, Dec. 1995.
- [21] A. Srivastava and A. Eustace. Atom - a system for building customized program analysis tools. In *Proc. of PLDI '94*, 1994.
- [22] D. Sweetman. *See MIPS Run*. Morgan Kaufman, 1999.
- [23] R. Wahbe, S. Lucco, T. Anderson, and S. Graham. Efficient software-based fault isolation. In *Proc. of the Fourteenth ACM SOSP*, pp. 203–216, Asheville, NC, USA, Dec. 1993.
- [24] D. Wall. Systems for late code modification. *CODE 91 Workshop on Code Generation*, 1991.
- [25] T. Wilkinson. Kaffe – a Java virtual machine. <http://www.transvirtual.com>.

A MIPS description

```
%{
char *directive = ".set noreorder\n.set nomacro\n"
                  ".set noat\n";
%}

regs = ( $0, $1, $2, $3, $4, $5, $6, $7, $8, $9,
          $10, $11, $12, $13, $14, $15, $16, $17,
          $18, $19, $20, $21, $22, $23, $24, $25,
          $26, $27, $28, $29, $30, $31 );

fregs = ( $f0, $f1, $f2, $f3, $f4, $f5, $f6, $f7,
           $f8, $f9, $f10, $f11, $f12, $f13, $f14,
           $f15, $f16, $f17, $f18, $f19, $f20,
           $f21, $f22, $f24, $f25, $f26, $f27, $f28,
           $f29, $f30, $f31 );

fcond = ( $fcc0, $fcc1, $fcc2, $fcc3, $fcc4, $fcc5,
           $fcc6, $fcc7 );

nop, sync, tlbr, tlbwi, tlbwr, tlbp, eret --> &op&;
movf, movt --> &op& r_d:regs, r_s:regs, r_c:fcond;
jr, jalr, mfhi, mthi, mflo, mtlo --> &op& r_s:regs;
jalr, tge, tgeu, tlt, tltu, teq, tne, mfc0, dmfc0,
cfc0, mtc0, dmtc0, ctc0, cfc1, mtc1, mfc2, cfc2,
mtc2, ctc2, mult, multu, dmult, dmultu
--> &op& r_d:regs, r_s:regs;
sll, sra, srl, dsll, dsrl, dsra, dsll32, dsrl32,
dsra32 --> &op& r_d:regs, r_w:regs, imm;
sllv, srav, srav, movz, movn, dsllv, dsrlv,
dsrav, add, addu, sub, subu, and, or, xor, nor,
slt, sltu, dadd, daddu, dsub, dsubu
--> &op& r_d:regs, r_w:regs, r_s:regs;

// MIPS assemblers use "div" as a macro
// this syntax is how the real hardware
// instructions of the same names can be generated
div, divu, ddiv, ddivu
--> &op& "$0", r_w:regs, r_s:regs;

break, syscall --> &op& imm;

j, jal --> &op &label&;

bltz, bgez, bltzl, bgezl, bltzal, bgezal, bltzall,
bgezall, bnezl, blezl, bgtzl, blez, bgtz
--> &op& r_s:regs, &ref& &label&;

tgei, tgeiu, tlti, tlти, teqi, tnei
--> &op& r_s:regs, imm;

beq, bne, beql, bnel
--> &op& r_s:regs, r_t:regs, &label&;

addi, addiu, slti, sltiu, andi, ori, xor, lui
--> &op& r_d:regs, &ref& imm;

daddi, daddiu --> &op& r_d:regs, r_w:regs, &ref& imm;

// we can define this form once and use it
// in different places (denoted by a single arrow)
```

```
branch -> &op& &label&;
branchc -> &op& r_s:fcond, &label&;

// these are coprocessor branches
bc0f, bc0t, bc0fl, bc0tl, bc2f, bc2t, bc2fl,
bc2tl: branch;

// these instructions come in two flavors
bcif, bcit, bcifl, bcitl: branch, branchc;

add.s, add.d, sub.s, sub.d, mul.s, mul.d, div.s,
div.d --> &op& r_d:fregs, r_w:fregs, r_s:fregs;

mfc1, dmfc1, dmfc1, ctcl
--> &op& r_t:regs, r_s:fregs;

sqrt.s, sqrt.d, abs.s, abs.d, mov.s, mov.d,
neg.s, neg.d, round.l.s, round.l.d, trunc.l.s,
trunc.l.d, ceil.l.s, ceil.l.d, floor.l.s,
floor.l.d, round.w.s, round.w.d, trunc.w.s,
trunc.w.d, ceil.w.s, ceil.w.d, floor.w.s,
floor.w.d, recip.s, recip.d, rsqrt.s, rsqrt.d,
cvt.s.d, cvt.s.w, cvt.s.l, cvt.d.s, cvt.d.w,
cvt.d.l, cvt.w.s, cvt.w.d, cvt.l.s, cvt.l.d
--> &op& r_d:fregs, r_s:fregs;

movf.s, movt.s, movf.d, movt.d
--> &op& r_d:fregs, r_w:fregs, r_s:fcond;

movz.s, movz.d, movn.s, movn.d
--> &op& r_d:fregs, r_w:fregs, r_s:regs;

c.f.s, c.f.d, c.un.s, c.un.d, c.eq.s, c.eq.d,
c.ueq.s, c.ueq.d, c.olts, c.oltd, c.ults,
c.ultd, c.ole.s, c.ole.d, c.ule.s, c.ule.d,
c.sf.s, c.sf.d, c.seq.s, c.seq.d, c.ngl.s,
c.ngl.d, c.lt.s, c.lt.d, c.nges, c.nged,
c.le.s, c.le.d, c.ngts, c.ngtd
--> &op& r_d:fcond, r_w:fregs, r_s:fregs;

// the assembler expects parentheses
lwxc1, ldxci, swxc1, sdxc1
--> &op& r_d:fregs, r_w:regs ( r_s:regs );

// prefetch instructions with hints
pref --> &op& r_h:hints, imm ( r_w:regs );
prefix --> &op& r_h:hints, r_w:regs ( r_s:regs );
hints = ( 0, 1, 4, 5, 6, 7 );

madd.s, madd.d, msub.s, msub.d, nmadd.s, nmadd.d,
nmsub.s, nmsub.d
--> &op& r_d:fregs, r_r:fregs, r_s:fregs, r_t:fregs;

ldl, ldr, lb, lh, lwl, lw, lbu, lhu, lwr, lwu, sb,
sh, swl, sw, sdl, sdr, swr, ll, lwc2, lld, ldc2, ld,
sc, swc2, scd, sdc2, sd
--> &op& r_d:regs, &ref& imm ( r_w:regs );

l.s, l.d, s.s, swc1, s.d, sdc1
--> &op& r_d:fregs, imm ( r_w:regs );

cache --> &op& r_d:cache_ops, imm ( r_w:regs );
cache_ops = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
               11, 13, 15, 16, 17, 18, 19, 20, 21,
               23, 24, 25, 27, 30, 31 );
```


An Embedded Error Recovery and Debugging Mechanism for Scripting Language Extensions

David M. Beazley

*Department of Computer Science
University of Chicago
Chicago, Illinois 60637
beazley@cs.uchicago.edu*

Abstract

In recent years, scripting languages such as Perl, Python, and Tcl have become popular development tools for the creation of sophisticated application software. One of the most useful features of these languages is their ability to easily interact with compiled languages such as C and C++. Although this mixed language approach has many benefits, one of the greatest drawbacks is the complexity of debugging that results from using interpreted and compiled code in the same application. In part, this is due to the fact that scripting language interpreters are unable to recover from catastrophic errors in compiled extension code. Moreover, traditional C/C++ debuggers do not provide a satisfactory degree of integration with interpreted languages. This paper describes an experimental system in which fatal extension errors such as segmentation faults, bus errors, and failed assertions are handled as scripting language exceptions. This system, which has been implemented as a general purpose shared library, requires no modifications to the target scripting language, introduces no performance penalty, and simplifies the debugging of mixed interpreted-compiled application software.

environments [7, 8, 9, 10, 11, 12, 13, 14, 15]. As a result, a large number of programmers are now using scripting languages to control complex C/C++ programs or as a tool for re-engineering legacy software. This approach is attractive because it allows programmers to benefit from the flexibility and rapid development of scripting while retaining the best features of compiled code such as high performance [16].

A critical aspect of scripting-compiled code integration is the way in which it departs from traditional C/C++ development and shell scripting. Rather than building stand-alone applications that run as separate processes, extension programming encourages a style of programming in which components are tightly integrated within an interpreter that is responsible for high-level control. Because of this, scripted software tends to rely heavily upon shared libraries, dynamic loading, scripts, and third-party extensions. In this sense, one might argue that the benefits of scripting are achieved at the expense of creating a more complicated development environment.

1 Introduction

Slightly more than ten years have passed since John Ousterhout introduced the Tcl scripting language at the 1990 USENIX technical conference [1]. Since then, scripting languages have been gaining in popularity as evidenced by the wide-spread use of systems such as Tcl, Perl, Python, Guile, PHP, and Ruby [1, 2, 3, 4, 5, 6].

In part, the success of modern scripting languages is due to their ability to be easily integrated with software written in compiled languages such as C, C++, and Fortran. In addition, a wide variety of wrapper generation tools can be used to automatically produce bindings between existing code and a variety of scripting language

A consequence of this complexity is an increased degree of difficulty associated with debugging programs that utilize multiple languages, dynamically loadable modules, and a sophisticated runtime environment. To address this problem, this paper describes an experimental system known as WAD (Wrapped Application Debugger) in which an embedded error reporting and debugging mechanism is added to common scripting languages. This system converts catastrophic signals such as segmentation faults and failed assertions to exceptions that can be handled by the scripting language interpreter. In doing so, it provides more seamless integration between error handling in scripting language interpreters and compiled extensions.

2 The Debugging Problem

Normally, a programming error in a scripted application results in an exception that describes the problem and the context in which it occurred. For example, an error in a Python script might produce a traceback similar to the following:

```
% python foo.py
Traceback (innermost last):
  File "foo.py", line 11, in ?
    foo()
  File "foo.py", line 8, in foo
    bar()
  File "foo.py", line 5, in bar
    spam()
  File "foo.py", line 2, in spam
    doh()
NameError: doh
```

In this case, a programmer might be able to apply a fix simply based on information in the traceback. Alternatively, if the problem is more complicated, a script-level debugger can be used to provide more information. In contrast, a failure in compiled extension code might produce the following result:

```
% python foo.py
Segmentation Fault (core dumped)
```

In this case, the user has no idea of what has happened other than it appears to be “very bad.” Furthermore, script-level debuggers are unable to identify the problem since they also crash when the error occurs (they run in the same process as the interpreter). This means that the only way for a user to narrow the source of the problem within a script is through trial-and-error techniques such as inserting print statements, commenting out sections of scripts, or having a deep intuition of the underlying implementation. Obviously, none of these techniques are particularly elegant.

An alternative approach is to run the application under the control of a traditional debugger such as gdb [17]. Although this provides some information about the error, the debugger mostly provides detailed information about the internal implementation of the scripting language interpreter instead of the script-level code that was running at the time of the error. Needless to say, this information isn’t very useful to most programmers. A related problem is that the structure of a scripted application tends to be much more complex than a traditional stand-alone program. As a result, a user may not have a good sense of how to actually attach an external debugger to their script. In addition, execution may occur within a complex run-time environment involving

events, threads, and network connections. Because of this, it can be difficult for the user to reproduce and identify certain types of catastrophic errors if they depend on timing or unusual event sequences. Finally, this approach requires a programmer to have a C development environment installed on their machine. Unfortunately, this may not hold in practice. This is because scripting languages are often used to provide programmability to applications where end-users write scripts, but do not write low-level C code.

Even if a traditional debugger such as gdb were modified to provide better integration with scripting languages, it is not clear that this would be the most natural solution to the problem. For one, having to run a separate debugging process to debug extension code is unnatural when no such requirement exists for scripts. Moreover, even if such a debugger existed, an inexperienced user may not have the expertise or inclination to use it. Finally, obscure fatal errors may occur long after an application has been deployed. Unless the debugger is distributed along with the application in some manner, it will be extraordinary difficult to obtain useful diagnostics when such errors occur.

The current state of the art in extension debugging is to simply add as much error checking as possible to extension modules. This is never a bad thing to do, but in practice it’s usually not enough to eliminate every possible problem. For one, scripting languages are sometimes used to control hundreds of thousands to millions of lines of compiled code. In this case, it is improbable that a programmer will foresee every conceivable error. In addition, scripting languages are often used to put new user interfaces on legacy software. In this case, scripting may introduce new modes of execution that cause a formerly “bug-free” application to fail in an unexpected manner. Finally, certain types of errors such as floating-point exceptions can be particularly difficult to eliminate because they might be generated algorithmically (e.g., as the result of instability in a numerical method). Therefore, even if a programmer has worked hard to eliminate crashes, there is usually a small probability that an application may fail under unusual circumstances.

3 Embedded Error Reporting

Rather than modifying an existing debugger to support scripting languages, an alternative approach is to add a more powerful error handling and reporting mechanism to the scripting language interpreter. We have implemented this approach in the form of an experimental system known as WAD. WAD is packaged as dynamically loadable shared library that can either be loaded as a scripting language extension module or linked to ex-

```

% python foo.py
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "foo.py", line 16, in ?
    foo()
  File "foo.py", line 13, in foo
    bar()
  File "foo.py", line 10, in bar
    spam()
  File "foo.py", line 7, in spam
    doh.doh(a,b,c)

SegFault: [ C stack trace ]

#2 0x00027774 in call_builtin(func=0x1c74f0,arg=0x1a1ccc,kw=0x0) in 'ceval.c',line 2650
#1 0xff083544 in _wrap_doh(self=0x0,args=0x1a1ccc) in 'foo_wrap.c',line 745
#0 0xfe7e0568 in doh(a=3,b=4,c=0x0) in 'foo.c',line 28

/u0/beazley/Projects/WAD/Python/foo.c, line 28

    int doh(int a, int b, int *c) {
=>    *c = a + b;
        return *c;
}

```

Figure 1: Cross language traceback generated by WAD for a segmentation fault in a Python extension

isting extension modules as a library. The core of the system is generic and requires no modifications to the scripting interpreter or existing extension modules. Furthermore, the system does not introduce a performance penalty as it does not rely upon program instrumentation or tracing.

WAD works by converting fatal signals such as SIGSEGV, SIGBUS, SIGFPE, and SIGABRT into scripting language exceptions that contain debugging information collected from the call-stack of compiled extension code. By handling errors in this manner, the scripting language interpreter is able to produce a cross-language stack trace that contains information from both the script code and extension code as shown for Python and Tcl/Tk in Figures 1 and 2. In this case, the user is given a very clear idea of what has happened without having to launch a separate debugger.

The advantage to this approach is that it provides more seamless integration between error handling in scripts and error handling in extensions. In addition, it eliminates the most common debugging step that a developer is likely to perform in the event of a fatal error—running a separate debugger on a core file and typing ‘where’ to get a stack trace. Finally, this allows end-users to provide extension writers with useful debugging information since they can supply a stack trace as opposed to a vague complaint that the program “crashed.”

4 Scripting Language Internals

In order to provide embedded error recovery, it is critical to understand how scripting language interpreters interface with extension code. Despite the wide variety of scripting languages, essentially every implementation uses a similar technique for accessing foreign code.

Virtually all scripting languages provide an extension mechanism in the form of a foreign function interface in which compiled procedures can be called from the scripting language interpreter. This is accomplished by writing a collection of wrapper functions that conform to a specified calling convention. The primary purpose of the wrappers are to marshal arguments and return values between the two languages and to handle errors. For example, in Tcl, every wrapper function must conform to the following prototype:

```

int
wrap_foo(ClientData clientData,
          Tcl_Interp *interp,
          int objc,
          Tcl_Obj *CONST objv[])
{
    /* Convert arguments */
    ...
    /* Call a function */

```

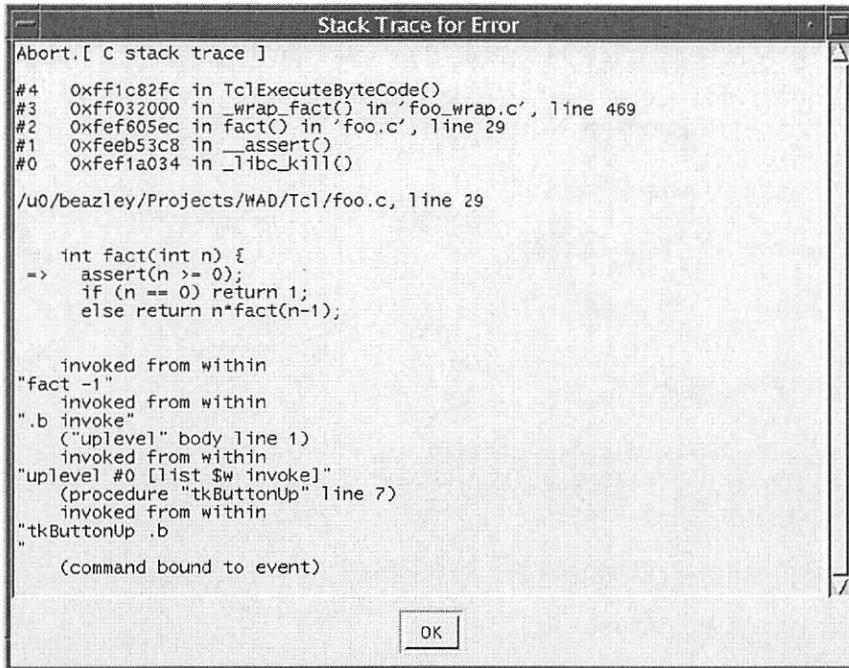


Figure 2: Dialog box with WAD generated traceback information for a failed assertion in a Tcl/Tk extension

```

result = foo(args);
/* Set result */
...
if (success) {
    return TCL_OK;
} else {
    return TCL_ERROR;
}
}

```

Another common extension mechanism is an object/type interface that allows programmers to create new kinds of fundamental types or attach special properties to objects in the interpreter. For example, both Tcl and Python provide an API for creating new “built-in” objects that behave like numbers, strings, lists, etc. In most cases, this involves setting up tables of function pointers that define various properties of an object. For example, if you wanted to add complex numbers to an interpreter, you might fill in a special data structure with pointers to methods that implement various numerical operations like this:

```

NumberMethods ComplexMethods {
    complex_add,
    complex_sub,
    complex_mul,
    complex_div,
    ...
}

```

Once registered with the interpreter, the methods in this structure would be invoked by various interpreter operators such as +, -, *, and /.

Most interpreters handle errors as a two-step process in which detailed error information is first registered with the interpreter and then a special error code is returned. For example, in Tcl, errors are handled by setting error information in the interpreter and returning a value of `TCL_ERROR`. Similarly in Python, errors are handled by calling a special function to raise an exception and returning `NULL`. In both cases, this triggers the interpreter’s error handler—possibly resulting in a stack trace of the running script. In some cases, an interpreter might handle errors using a form of the C `longjmp` function. For example, Perl provides a special function `die` that jumps back to the interpreter with a fatal error [11].

The precise implementation details of these mechanisms aren’t so important for our discussion. The critical point is that scripting languages always access extension code through a well-defined interface that precisely defines how arguments are to be passed, values are to be returned, and errors are to be handled.

5 Scripting Languages and Signals

Under normal circumstances, errors in extension code are handled through the error-handling API provided by the scripting language interpreter. For example, if an

invalid function parameter is passed, a program can simply set an error message and return to the interpreter. Similarly, automatic wrapper generators such as SWIG can produce code to convert C++ exceptions and other C-related error handling schemes to scripting language errors [18]. On the other hand, segmentation faults, failed assertions, and similar problems produce signals that cause the interpreter to abort execution.

Most scripting languages provide limited support for Unix signal handling [19]. However, this support is not sufficiently advanced to recover from fatal signals produced by extension code. Unlike signals generated for asynchronous events such as I/O, execution can *not* be resumed at the point of a fatal signal. Therefore, even if such a signal could be caught and handled by a script, there isn't much that it can do except to print a diagnostic message and abort before the signal handler returns. In addition, some interpreters block signal delivery while executing extension code—opting to handle signals at a time when it is more convenient. In this case, a signal such as SIGSEGV would simply cause the whole application to freeze since there is no way for execution to continue to a point where the signal could be delivered. Thus, scripting languages tend to either ignore the problem or label it as a “limitation.”

6 Overview of WAD

WAD installs a signal handler for SIGSEGV, SIGBUS, SIGABRT, SIGILL, and SIGFPE using the `sigaction` function [19]. Furthermore, it uses a special option (`SA_SIGINFO`) of signal handling that passes process context information to the signal handler when a signal occurs. Since none of these signals are normally used in the implementation of the scripting interpreter or by user scripts, this does not usually override any previous signal handling. Afterwards, when one of these signals occurs, a two-phase recovery process executes. First, information is collected about the execution context including a full stack-trace, symbol table entries, and debugging information. Then, the current stream of execution is aborted and an error is returned to the interpreter. This process is illustrated in Figure 3.

The collection of context and debugging information involves the following steps:

- The program counter and stack pointer are obtained from context information passed to the signal handler.
- The virtual memory map of the process is obtained from /proc and used to associate virtual memory addresses with executable files, shared libraries, and dynamically loaded extension modules [20].

- The call stack is unwound to collect traceback information. At each step of the stack traceback, symbol table and debugging information is gathered and stored in a generic data structure for later use in the recovery process. This data is obtained by memory-mapping the object files associated with the process and extracting symbol table and debugging information.

Once debugging information has been collected, the signal handler enters an error-recovery phase that attempts to raise a scripting exception and return to a suitable location in the interpreter. To do this, the following steps are performed:

- The stack trace is examined to see if there are any locations in the interpreter to which control can be returned.
- If a suitable return location is found, the CPU context is modified in a manner that makes the signal handler return to the interpreter with an error. This return process is assisted by a small trampoline function (partially written in assembly language) that arranges a proper return to the interpreter after the signal handler returns.

Of the two phases, the first is the most straightforward to implement because it involves standard Unix API functions and common file formats such as ELF and stabs [21, 22]. On the other hand, the recovery phase in which control is returned to the interpreter is of greater interest. Therefore, it is now described in greater detail.

7 Returning to the Interpreter

To return to the interpreter, WAD maintains a table of symbolic names that correspond to locations within the interpreter responsible for invoking wrapper functions and object/type methods. For example, Table 1 shows a partial list of return locations used in the Python implementation. When an error occurs, the call stack is scanned for the first occurrence of any symbol in this table. If a match is found, control is returned to that location by emulating the return of a wrapper function with the error code from the table. If no match is found, the error handler simply prints a stack trace to standard output and aborts.

When a symbolic match is found, WAD invokes a special user-defined handler function that is written for a specific scripting language. The primary role of this handler is to take debugging information gathered from the call stack and generate an appropriate scripting language error. One peculiar problem of this step is that the

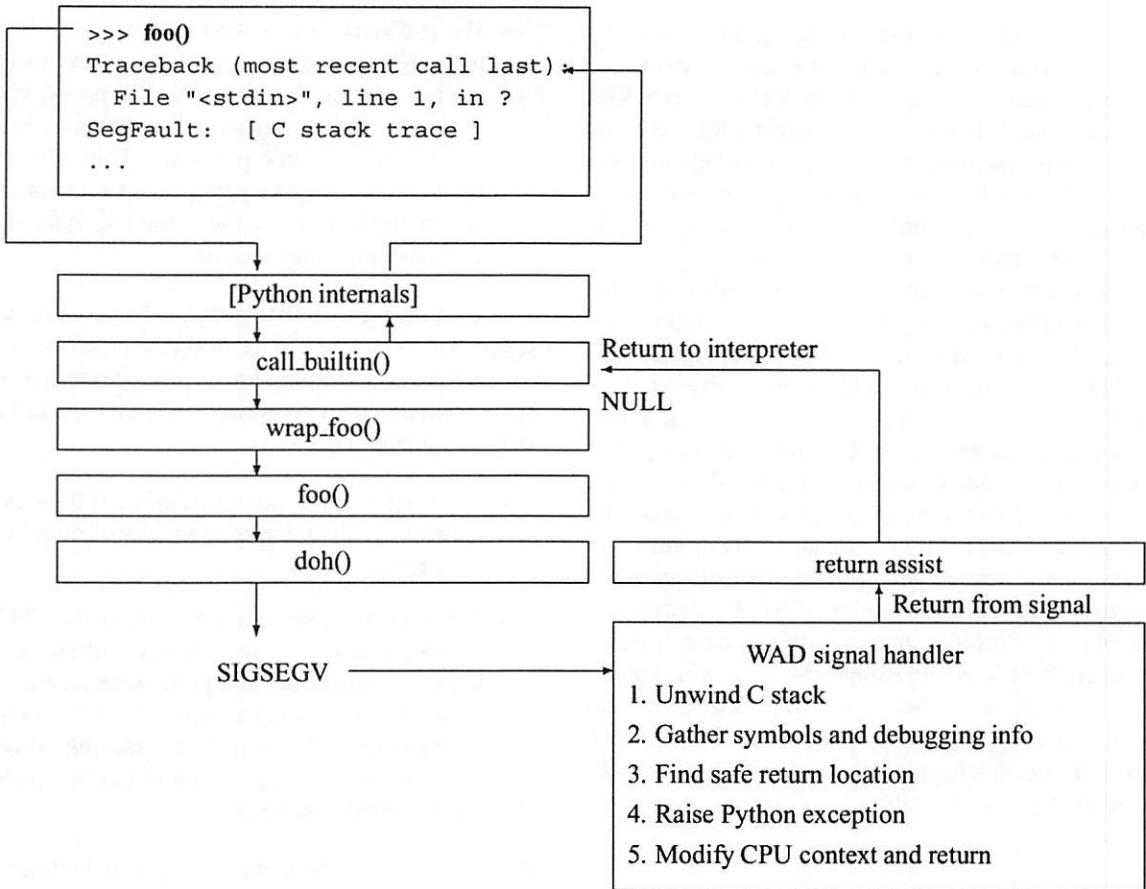


Figure 3: Control Flow of the Error Recovery Mechanism for Python

generation of an error may require the use of parameters passed to a wrapper function. For example, in the Tcl wrapper shown earlier, one of the arguments was an object of type “`Tcl_Interp *`”. This object contains information specific to the state of the interpreter (and multiple interpreter objects may exist in a single application). Unfortunately, no reference to the interpreter object is available in the signal handler nor is a reference to interpreter guaranteed to exist in the context of a function that generated the error.

To work around this problem, WAD implements a feature known as argument stealing. When examining the call-stack, the signal handler has full access to all function arguments and local variables of each function on the stack. Therefore, if the handler knows that an error was generated while calling a wrapper function (as determined by looking at the symbol names), it can grab the interpreter object from the stack frame of the wrapper and use it to set an appropriate error code before returning to the interpreter. Currently, this is managed by allowing the signal handler to steal arguments from the caller using positional information. For example, to grab

the `Tcl_Interp *` object from a Tcl wrapper function, code similar to the following is written:

```

Tcl_Interp *interp;
int err;

interp = (Tcl_Interp *)
    wad_stole_outarg(
        stack,
        "TclExecuteByteCode",
        1,
        &err
    );
...
if (!err) {
    Tcl_SetResult(interp, errtype, ...);
    Tcl_AddErrorInfo(interp, errdetails);
}

```

In this case, the Tcl interpreter argument passed to a wrapper function is stolen and used to generate an error. Also, the name `TclExecuteByteCode` refers to the calling function, not the wrapper function itself. At

Python symbol	Error return value
call_builtin	NULL
PyObject_Print	-1
PyObject_CallFunction	NULL
PyObject_CallMethod	NULL
PyObject_CallObject	NULL
PyObject_Cmp	-1
PyObject_DelAttrString	-1
PyObject_DelItem	-1
PyObject_GetAttrString	NULL

Table 1: A partial list of symbolic return locations in the Python interpreter

this time, argument stealing is only applicable to simple types such as integers and pointers. However, this appears to be adequate for generating scripting language errors.

8 Register Management

A final issue concerning the return mechanism has to do with the behavior of the non-local return to the interpreter. Roughly speaking, this emulates the C `longjmp` library call. However, this is done without the use of a matching `setjmp` in the interpreter.

The primary problem with aborting execution and returning to the interpreter in this manner is that most compilers use a register management technique known as callee-save [23]. In this case, it is the responsibility of the called function to save the state of the registers and to restore them before returning to the caller. By making a non-local jump, registers may be left in an inconsistent state due to the fact that they are not restored to their original values. The `longjmp` function in the C library avoids this problem by relying upon `setjmp` to save the registers. Unfortunately, WAD does not have this luxury. As a result, a return from the signal handler may produce a corrupted set of registers at the point of return in the interpreter.

The severity of this problem depends greatly on the architecture and compiler. For example, on the SPARC, register windows effectively solve the callee-save problem [24]. In this case, each stack frame has its own register window and the windows are flushed to the stack whenever a signal occurs. Therefore, the recovery mechanism can simply examine the stack and arrange to restore the registers to their proper values when control is returned. Furthermore, certain conventions of the SPARC ABI resolve several related issues. For example, floating point registers are caller-saved and the con-

tents of the SPARC global registers are not guaranteed to be preserved across procedure calls (in fact, they are not even saved by `setjmp`).

On other platforms, the problem of register management becomes more interesting. In this case, a heuristic approach that examines the machine code for each function on the call stack can be used to determine where the registers might have been saved. This approach is used by `gdb` and other debuggers when they allow users to inspect register values within arbitrary stack frames [17]. Even though this sounds complicated to implement, the algorithm is greatly simplified by the fact that compilers typically generate code to store the callee-save registers immediately upon the entry to each function. In addition, this code is highly regular and easy to examine. For instance, on i386-Linux, the callee-save registers can be restored by simply examining the first few bytes of the machine code for each function on the call stack to figure out where values have been saved. The following code shows a typical sequence of machine instructions used to store callee-save registers on i386-Linux:

```
foo:
55      pushl %ebp
89 e5    mov %esp, %ebp
83 a0    subl $0xa0,%esp
56      pushl %esi
57      pushl %edi
...
```

As a fall-back, WAD could be configured to return control to a location previously specified with `setjmp`. Unfortunately, this either requires modifications to the interpreter or its extension modules. Although this kind of instrumentation could be facilitated by automatic wrapper code generators, it is not a preferred solution and is not discussed further.

9 Initialization

To simplify the debugging of extension modules, it is desirable to make the use of WAD as transparent as possible. Currently, there are two ways in which the system is used. First, WAD may be explicitly loaded as a scripting language extension module. For instance, in Python, a user can include the statement `import libwadpy` in a script to load the debugger. Alternatively, WAD can be enabled by linking it to an extension module as a shared library. For instance:

```
% ld -shared $(OBJS) -lwadpy
```

In this latter case, WAD initializes itself whenever the extension module is loaded. The same shared library

is used for both situations by making sure two types of initialization techniques are used. First, an empty initialization function is written to make WAD appear like a proper scripting language extension module (although it adds no functions to the interpreter). Second, the real initialization of the system is placed into the initialization section of the WAD shared library object file (the “init” section of ELF files). This code always executes when a library is loaded by the dynamic loader is commonly used to properly initialize C++ objects. Therefore, a fairly portable way to force code into the initialization section is to encapsulate the initialization in a C++ statically constructed object like this:

```
class InitWad {
public:
    InitWad() { wad_init(); }
};

/* This forces InitWad() to execute
   on loading. */
static InitWad init;
```

The nice part about this technique is that it allows WAD to be enabled simply by linking or loading; no special initialization code needs to be added to an extension module to make it work. In addition, due to the way in which the loader resolves and initializes libraries, the initialization of WAD is guaranteed to execute before any of the code in the extension module to which it has been linked. The primary downside to this approach is that the WAD shared object file can not be linked directly to an interpreter. This is because WAD sometimes needs to call the interpreter to properly initialize its exception handling mechanism (for instance, in Python, four new types of exceptions are added to the interpreter). Clearly this type of initialization is impossible if WAD is linked directly to an interpreter as its initialization process would execute before the main program of the interpreter started. However, if you wanted to permanently add WAD to an interpreter, the problem is easily corrected by first removing the C++ initializer from WAD and then replacing it with an explicit initialization call someplace within the interpreter’s startup function.

10 Exception Objects

Before WAD returns control to the interpreter, it collects all of the stack-trace and debugging information it was able to obtain into a special exception object. This object represents the state of the call stack and includes things like symbolic names for each stack frame, the names, types, and values of function parameters and stack variables, as well as a complete copy of data on the stack. This information is represented in a generic

manner that hides platform specific details related to the CPU, object file formats, debugging tables, and so forth.

Minimally, the exception data is used to print a stack trace as shown in Figure 1. However, if the interpreter is successfully able to regain control, the contents of the exception object can be freely examined after an error has occurred. For example, a Python script could catch a segmentation fault and print debugging information like this:

```
try:
    # Some buggy code
    ...
except SegFault,e:
    print 'Whoa!'
    # Get WAD exception object
    t = e.args[0]
    # Print location info
    print t.__FILE__
    print t.__LINE__
    print t.__NAME__
    print t.__SOURCE__
    ...
    ...
```

Inspection of the exception object also makes it possible to write post mortem script debuggers that merge the call stacks of the two languages and provide cross language diagnostics. Figure 4 shows an example of a simple mixed language debugging session using the WAD post-mortem debugger (wpm) after an extension error has occurred in a Python program. In the figure, the user is first presented with a multi-language stack trace. The information in this trace is obtained both from the WAD exception object and from the Python traceback generated when the exception was raised. Next, we see the user walking up the call stack using the ‘u’ command of the debugger. As this proceeds, there is a seamless transition from C to Python where the trace crosses between the two languages. An optional feature of the debugger (not shown) allows the debugger to walk up the entire C call-stack (in this case, the trace shows information about the implementation of the Python interpreter). More advanced features of the debugger allow the user to query values of function parameters, local variables, and stack frames (although some of this information may not be obtainable due to compiler optimizations and the difficulties of accurately recovering register values).

11 Implementation Details

Currently, WAD is implemented in ANSI C and small amount of assembly code to assist in the return to the interpreter. The current implementation supports

```

[ Error occurred ]
>>> from wpm import *
*** WAD Debugger ***
#5  [ Python ] in self.widget._report_exception() in ...
#4  [ Python ] in Button(self,text="Die", command=lambda x=self: ...
#3  [ Python ] in death_by_segmentation() in death.py, line 22
#2  [ Python ] in debug(seg_crash()) in death.py, line 5
#1  0xfee2780 in _wrap_seg_crash(self=0x0,args=0x18f114) in 'pydebug.c', line 512
#0  0xee1320 in seg_crash() in 'debug.c', line 20

    int *a = 0;
=>  *a = 3;
    return 1;

>>> u
#1  0xfee2780 in _wrap_seg_crash(self=0x0,args=0x18f114) in 'pydebug.c', line 512

    if(!PyArg_ParseTuple(args,:seg_crash)) return NULL;
=>  result = (int )seg_crash();
    resultobj = PyInt_FromLong((long)result);

>>> u
#2  [ Python ] in debug(seg_crash()) in death.py, line 5

    def death_by_segmentation():
=>      debug(seg_crash())

>>> u
#3  [ Python ] in death_by_segmentation() in death.py, line 22

    if ty == 1:
=>      death_by_segmentation()
    elif ty == 2:
>>>

```

Figure 4: Cross-language debugging session in Python where a user is walking a mixed language call stack.

Python and Tcl extensions on SPARC Solaris and i386-Linux. Each scripting language is currently supported by a separate shared library such as `libwadpy.so` and `libwadtcl.so`. In addition, a language neutral library `libwad.so` can be linked against non-scripted applications (in which case a stack trace is simply printed to standard error when a problem occurs). The entire implementation contains approximately 2000 semicolons. Most of this code pertains to the gathering of debugging information from object files. Only a small part of the code is specific to a particular scripting language (170 semicolons for Python and 50 semicolons for Tcl).

Although there are libraries such as the GNU Binary File Descriptor (BFD) library that can assist with the manipulation of object files, these are not used in the implementation [25]. These libraries tend to be quite large and are oriented more towards stand-alone tools such as de-

buggers, linkers, and loaders. In addition, the behavior of these libraries with respect to memory management would need to be carefully studied before they could be safely used in an embedded environment. Finally, given the small size of the prototype implementation, it didn't seem necessary to rely upon such a heavyweight solution.

A surprising feature of the implementation is that a significant amount of the code is language independent. This is achieved by placing all of the process introspection, data collection, and platform specific code within a centralized core. To provide a specific scripting language interface, a developer only needs to supply two things; a table containing symbolic function names where control can be returned (Table 1), and a handler function in the form of a callback. As input, this handler receives an exception object as described in an earlier section. From this, the handler can raise a scripting

language exception in whatever manner is most appropriate.

Significant portions of the core are also relatively straightforward to port between different Unix systems. For instance, code to read ELF object files and stabs debugging data is essentially identical for Linux and Solaris. In addition, the high-level control logic is unchanged between platforms. Platform specific differences primarily arise in the obvious places such as the examination of CPU registers, manipulation of the process context in the signal handler, reading virtual memory maps from /proc, and so forth. Additional changes would also need to be made on systems with different object file formats such as COFF and DWARF2. To extent that it is possible, these differences could be hidden by abstraction mechanisms (although the initial implementation of WAD is weak in this regard and would benefit from techniques used in more advanced debuggers such as gdb). Despite these porting issues, the primary requirement for WAD is a fully functional implementation of SVR4 signal handling that allows for modifications of the process context.

Due to the heavy dependence on Unix signal handling, process introspection, and object file formats, it is unlikely that WAD could be easily ported to non-Unix systems such as Windows. However, it may be possible to provide a similar capability using advanced features of Windows structured exception handling [26]. For instance, structured exception handlers can be used to catch hardware faults, they can receive process context information, and they can arrange to take corrective action much like the signal implementation described here.

12 Modification of Interpreters?

A logical question to ask about the implementation of WAD is whether or not it would make sense to modify existing interpreters to assist in the recovery process. For instance, instrumenting Python or Tcl with setjmp functions might simplify the implementation since it would eliminate issues related to register restoration and finding a suitable return location.

Although it may be possible to make these changes, there are several drawbacks to this approach. First, the number of required modifications may be quite large. For instance, there are well over 50 entry points to extension code within the implementation of Python. Second, an extension module may perform callbacks and evaluation of script code. This means that the call stack would cross back and forth between languages and that these modifications would have to be made in a way that allows arbitrary nesting of extension calls. Finally, instrumenting the code in this manner may introduce a perfor-

mance impact—a clearly undesirable side effect considering the infrequent occurrence of fatal extension errors.

13 Discussion

The primary goal of embedded error recovery is to provide an alternative approach for debugging scripting language extensions. Although this approach has many benefits, there are a number drawbacks and issues that must be discussed.

First, like the C longjmp function, the error recovery mechanism does not cleanly unwind the call stack. For C++, this means that objects allocated on stack will not be finalized (destructors will not be invoked) and that memory allocated on the heap may be leaked. Similarly, this could result in open files, sockets, and other system resources. In a multi-threaded environment, deadlock may occur if a procedure holds a lock when an error occurs.

In certain cases, the use of signals in WAD may interact adversely with scripting language signal handling. Since scripting languages ordinarily do not catch signals such as SIGSEGV, SIGBUS, and SIGABRT, the use of WAD is unlikely to conflict with any existing signal handling. However, most scripting languages would not prevent a user from disabling the WAD error recovery mechanism by simply specifying a new handler for one or more of these signals. In addition, the use of certain extensions such as the Perl sigtrap module would completely disable WAD [2].

A more difficult signal handling problem arises when thread libraries are used. These libraries tend to override default signal handling behavior in a way that defines how signals are delivered to each thread [27]. In general, asynchronous signals can be delivered to any thread within a process. However, this does not appear to be a problem for WAD since hardware exceptions are delivered to a signal handler that runs within the same thread in which the error occurred. Unfortunately, even in this case, personal experience has shown that certain implementations of user thread libraries (particularly on older versions of Linux) do not reliably pass signal context information nor do they universally support advanced signal operations such as sigaltstack. Because of this, WAD may be incompatible with a crippled implementation of user threads on these platforms.

A even more subtle problem with threads is that the recovery process itself is not thread-safe (i.e., it is not possible to concurrently handle fatal errors occurring in different threads). For most scripting language extensions, this limitation does not apply due to strict run-time restrictions that interpreters currently place on thread support. For instance, even though Python supports

threaded programs, it places a global mutex-lock around the interpreter that makes it impossible for more than one thread to concurrently execute within the interpreter at once. A consequence of this restriction is that extension functions are not interruptible by thread-switching unless they explicitly release the interpreter lock. Currently, the behavior of WAD is undefined if extension code releases the lock and proceeds to generate a fault. In this case, the recovery process may either cause an exception to be raised in an entirely different thread or cause execution to violate the interpreter's mutual exclusion constraint on the interpreter.

In certain cases, errors may result in an unrecoverable crash. For example, if an application overwrites the heap, it may destroy critical data structures within the interpreter. Similarly, destruction of the call stack (via buffer overflow) makes it impossible for the recovery mechanism to create a stack-trace and return to the interpreter. More subtle memory management problems such as double-freeing of heap allocated memory can also cause a system to fail in a manner that bears little resemblance to actual source of the problem. Given that WAD lives in the same process as the faulting application and that such errors may occur, a common question to ask is to what extent does WAD complicate debugging when it doesn't work.

To handle potential problems in the implementation of WAD itself, great care is taken to avoid the use of library functions and functions that rely on heap allocation (`malloc`, `free`, etc.). For instance, to provide dynamic memory allocation, WAD implements its own memory allocator using `mmap`. In addition, signals are disabled immediately upon entry to the WAD signal handler. Should a fatal error occur inside WAD, the application will dump core and exit. Since the resulting core file contains the stack trace of both WAD and the faulting application, a traditional C debugger can be used to identify the problem as before. The only difference is that a few additional stack frames will appear on the traceback.

An application may also fail after the WAD signal handler has completed execution if memory or stack frames within the interpreter have been corrupted in a way that prevents proper exception handling. In this case, the application may fail in a manner that does not represent the original programming error. It might also cause the WAD signal handler to be immediately reinvoked with a different process state—causing it to report information about a different type of failure. To address these kinds of problems, WAD creates a tracefile `wadtrace` in the current working directory that contains information about each error that it has handled. If no recovery was possible, a programmer can look at this file to obtain all of the stack traces that were generated.

If an application is experiencing a very serious prob-

lem, WAD does not prevent a standard debugger from being attached to the process. This is because the debugger overrides the current signal handling so that it can catch fatal errors. As a result, even if WAD is loaded, fatal signals are simply redirected to the attached debugger. Such an approach also allows for more complex debugging tasks such as single-step execution, breakpoints, and watchpoints—none of which are easily added to WAD itself.

Finally, there are a number of issues that pertain to the interaction of the recovery mechanism with the interpreter. For instance, the recovery scheme is unable to return to procedures that might invoke wrapper functions with conflicting return codes. This problem manifests itself when the interpreter's virtual machine is built around a large `switch` statement from which different types of wrapper functions are called. For example, in Python, certain internal procedures call a mix of functions where both `NULL` and `-1` are returned to indicate errors (depending on the function). In this case, there is no way to specify a proper error return value because there will be conflicting entries in the WAD return table (although you could compromise and return the error value for the most common case). The recovery process is also extremely inefficient due to its heavy reliance on `mmap`, file I/O, and linear search algorithms for finding symbols and debugging information. Therefore, WAD would unsuitable as a more general purpose extension related exception handler.

Despite these limitations, embedded error recovery is still a useful capability that can be applied to a wide variety of extension related errors. This is because errors such as failed assertions, bus errors, and floating point exceptions rarely result in a situation where the recovery process would be unable to run or the interpreter would crash. Furthermore, more serious errors such as segmentation faults are more likely to be caused by an uninitialized pointer than a blatant destruction of the heap or stack.

14 Related Work

A huge body of literature is devoted to the topic of exception handling in various languages and systems. Furthermore, the topic remains one of active interest in the software community. For instance, IEEE Transactions on Software Engineering recently devoted two entire issues to current trends in exception handling [29, 30]. Unfortunately, very little of this work seems to be directly related to mixed compiled-interpreted exception handling, recovery from fatal signals, and problems pertaining to mixed-language debugging.

Perhaps the most directly relevant work is that of advanced programming environments for Common Lisp

[31]. Not only does CL have a foreign function interface, debuggers such as gdb have previously been modified to walk the Lisp stack [33, 34]. Furthermore, certain Lisp development environments have previously provided a high degree of integration between compiled code and the Lisp interpreter[32].

In certain cases, a scripting language module has been used to provide partial information for fatal signals. For example, the Perl `sigtrap` module can be used to produce a Perl stack trace when a problem occurs [2]. Unfortunately, this module does not provide any information from the C stack. Similarly, advanced software development environments such as Microsoft's Visual Studio can automatically launch a C/C++ debugger when an error occurs. Unfortunately, this doesn't provide any information about the script that was running.

In the area of programming languages, a number of efforts have been made to map signals to exceptions in the form of asynchronous exception handling [35, 37, 36]. Unfortunately, this work tends to concentrate on the problem of handling asynchronous signals related to I/O as opposed to synchronously generated signals caused by software faults.

With respect to debugging, little work appears to have been done in the area of mixed compiled-interpreted debugging. Although modern debuggers certainly try to provide advanced capabilities for debugging within a single language, they tend to ignore the boundary between languages. As previously mentioned, debuggers have occasionally been modified to support other languages such as Common Lisp [34]. However, little work appears to have been done in the context of modern scripting languages. One system of possible interest in the context of mixed compiled-interpreted debugging is the *Rⁿ* system developed at Rice University in the mid-1980's [38]. This system, primarily developed for scientific computing, allowed control to transparently pass between compiled code and an interpreter. Furthermore, the system allowed dynamic patching of an executable in which compiled procedures could be replaced by an interpreted replacement. Although this system does not directly pertain to the problem of debugging of scripting language extensions, it is one of the few examples of a system in which compiled and interpreted code have been tightly integrated within a debugger.

More recently, a couple of efforts have emerged to that seem to address certain issues related to mixed-mode debugging of interpreted and compiled code. PyDebug is a recently developed system that focuses on problems related to the management of breakpoints in Python extension code [39]. It may also be possible to perform mixed-mode debugging of Java and native methods using features of the Java Platform Debugger Architecture (JPDA) [40]. Mixed-mode debugging support for Java

may also be supported in advanced debugging systems such as ICAT [41]. However, none of these systems appear to have taken the approach of converting hardware faults into Java errors or exceptions.

15 Future Directions

As of this writing, WAD is only an experimental prototype. Because of this, there are certainly a wide variety of incremental improvements that could be made to support additional platforms and scripting languages. In addition, there are a variety of improvements that could be made to provide better integration with threads and C++. One could also investigate heuristic schemes such as backward stack tracing that might be able to recover partial debugging information from corrupted call stacks [28].

A more interesting extension of this work would be to see how the exception handling approach of WAD could be incorporated with the integrated development environments and script-level debugging systems that have already been developed. For instance, it would be interesting to see if a graphical debugging front-end such as DDD could be modified to handle mixed-language stack traces within the context of a script-level debugger [42].

It may also be possible to extend the approach taken by WAD to other types of extensible systems. For instance, if one were developing a new server module for the Apache web-server, it might be possible to redirect fatal module errors back to the server in a way that produces a webpage with a stack trace [43]. The exception handling approach may also have applicability to situations where compiled code is used to build software components that are used as part of a large distributed system.

16 Conclusions and Availability

This paper has presented a mechanism by which fatal errors such as segmentation faults and failed assertions can be handled as scripting language exceptions. This approach, which relies upon advanced features of Unix signal handling, allows fatal signals to be caught and transformed into errors from which interpreters can produce an informative cross-language stack trace. In doing so, it provides more seamless integration between scripting languages and compiled extensions. Furthermore, this has the potential to greatly simplify the frustrating task of debugging complicated mixed scripted-compiled software.

The prototype implementation of this system is available at :

<http://systems.cs.uchicago.edu/wad>.

Currently, WAD supports Python and Tcl on SPARC Solaris and i386-Linux systems. Work to support additional scripting languages and platforms is ongoing.

17 Acknowledgments

Richard Gabriel and Harlan Sexton provided interesting insights concerning debugging capabilities in Common Lisp. Stephen Hahn provided useful information concerning the low-level details of signal handling on Solaris. I would also like to thank the technical reviewers and Rob Miller for their useful comments.

References

- [1] J. K. Ousterhout, *Tcl: An Embeddable Command Language*, Proceedings of the USENIX Association Winter Conference, 1990. p.133-146.
- [2] L. Wall, T. Christiansen, and R. Schwartz, *Programming Perl*, 2nd. Ed. O'Reilly & Associates, 1996.
- [3] M. Lutz, *Programming Python*, O'Reilly & Associates, 1996.
- [4] Thomas Lord, *An Anatomy of Guile, The Interface to Tcl/Tk*, USENIX 3rd Annual Tcl/Tk Workshop 1995.
- [5] T. Ratschiller and T. Gerken, *Web Application Development with PHP 4.0*, New Riders, 2000.
- [6] D. Thomas, A. Hunt, *Programming Ruby*, Addison-Wesley, 2001.
- [7] D.M. Beazley, *SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++*, Proceedings of the 4th USENIX Tcl/Tk Workshop, p. 129-139, July 1996.
- [8] P. Thompson, *SIP*,
<http://www.thekompany.com/projects/pykde>.
- [9] P. F. Dubois, *Climate Data Analysis Software*, 8th International Python Conference, Arlington, VA., 2000.
- [10] P. Peterson, J. Martins, and J. Alonso, *Fortran to Python Interface Generator with an application to Aerospace Engineering*, 9th International Python Conference, submitted, 2000.
- [11] S. Srinivasan, *Advanced Perl Programming*, O'Reilly & Associates, 1997.
- [12] Wolfgang Heidrich and Philipp Slusallek, *Automatic Generation of Tcl Bindings for C and C++ Libraries*, USENIX 3rd Tcl/Tk Workshop, 1995.
- [13] K. Martin, *Automated Wrapping of a C++ Class Library into Tcl*, USENIX 4th Tcl/Tk Workshop, p. 141-148, 1996.
- [14] C. Lee, *G-Wrap: A tool for exporting C libraries into Scheme Interpreters*,
<http://www.cs.cmu.edu/~chrislee/Software/g-wrap>.
- [15] G. Couch, C. Huang, and T. Ferrin, *Wrappy :A Python Wrapper Generator for C++ Classes*, O'Reilly Open Source Software Convention, 1999.
- [16] J. K. Ousterhout, *Scripting: Higher-Level Programming for the 21st Century*, IEEE Computer, Vol 31, No. 3, p. 23-30, 1998.
- [17] R. Stallman and R. Pesch, *Using GDB: A Guide to the GNU Source-Level Debugger*. Free Software Foundation and Cygnus Support, Cambridge, MA, 1991.
- [18] D.M. Beazley and P.S. Lomdahl, *Feeding a Large-scale Physics Application to Python*, 6th International Python Conference, co-sponsored by USENIX, p. 21-28, 1997.
- [19] W. Richard Stevens, *UNIX Network Programming: Interprocess Communication, Volume 2*. PTR Prentice-Hall, 1998.
- [20] R. Faulkner and R. Gomes, *The Process File System and Process Model in UNIX System V*, USENIX Conference Proceedings, January 1991.
- [21] J. R. Levine, *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.
- [22] Free Software Foundation, *The "stabs" debugging format*. GNU info document.
- [23] M.L. Scott. *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2000.
- [24] D. Weaver and T. Germond, *SPARC Architecture Manual Version 9*, Prentice-Hall, 1993.
- [25] S. Chamberlain. *libbfd: The Binary File Descriptor Library*. Cygnus Support, bfd version 3.0 edition, April 1991.

- [26] M. Pietrek, *A Crash Course on the Depths of Win32 Structured Exception Handling*, Microsoft Systems Journal, January 1997.
- [27] F. Mueller, *A Library Implementation of POSIX Threads Under Unix*, USENIX Winter Technical Conference, San Diego, CA., p. 29-42, 1993.
- [28] J. B. Rosenberg, *How Debuggers Work: Algorithms, Data Structures, and Architecture*, John Wiley & Sons, 1996.
- [29] D.E. Perry, A. Romanovsky, and A. Tripathi, *Current Trends in Exception Handling-Part I*, IEEE Transactions on Software Engineering, Vol 26, No. 9, p. 817-819, 2000.
- [30] D.E. Perry, A. Romanovsky, and A. Tripathi, *Current Trends in Exception Handling-Part II*, IEEE Transactions on Software Engineering, Vol 26, No. 10, p. 921-922, 2000.
- [31] G.L. Steele Jr., *Common Lisp: The Language, Second Edition*, Digital Press, Bedford, MA. 1990.
- [32] R. Gabriel, private correspondence.
- [33] H. Sexton, *Foreign Functions and Common Lisp*, in Lisp Pointers, Vol 1, No. 5, 1988.
- [34] W. Henessey, *WCL: Delivering Efficient Common Lisp Applications Under Unix*, ACM Conference on Lisp and Functional Languages, p. 260-269, 1992.
- [35] P.A. Buhr and W.Y.R. Mok, *Advanced Exception Handling Mechanisms*, IEEE Transactions on Software Engineering, Vol. 26, No. 9, p. 820-836, 2000.
- [36] S. Marlow, S. P. Jones, and A. Moran. *Asynchronous Exceptions in Haskell*. In 4th International Workshop on High-Level Concurrent Languages, September 2000.
- [37] J. H. Reppy, *Asynchronous Signals in Standard ML*. Technical Report TR90-1144, Cornell University, Computer Science Department, 1990.
- [38] A. Carle, D. Cooper, R. Hood, K. Kennedy, L. Tarczon, S. Warren, *A Practical Environment for Scientific Programming*. IEEE Computer, Vol 20, No. 11, p. 75-89, 1987.
- [39] P. Stoltz, *PyDebug, a New Application for Integrated Debugging of Python with C and Fortran Extensions*, O'Reilly Open Source Software Convention, San Diego, 2001 (to appear).
- [40] Sun Microsystems, *Java Platform Debugger Architecture*, <http://java.sun.com/products/jpda>
- [41] IBM, *ICAT Debugger*, <http://techsupport.services.ibm.com/icat>.
- [42] A. Zeller, *Visual Debugging with DDD*, Dr. Dobb's Journal, March, 2001.
- [43] Apache HTTP Server Project, <http://httpd.apache.org/>

Interactive Simultaneous Editing of Multiple Text Regions

Robert C. Miller and Brad A. Myers

Carnegie Mellon University

<http://www.cs.cmu.edu/~rcm/lapis/>

{rcm, bam}@cs.cmu.edu

Abstract

Simultaneous editing is a new method for automating repetitive text editing. After describing a set of regions to edit (the *records*), the user can edit any one record and see equivalent edits applied simultaneously to all other records. The essence of simultaneous editing is generalizing the user's selection in one record to equivalent selections in the other records. We describe a generalization method that is fast (suitable for interactive use), domain-specific (capable of using high-level knowledge such as Java and HTML syntax), and under user control (generalizations can be corrected or overridden). Simultaneous editing is useful for source code editing, HTML editing, and scripting, as well as many other applications.

1 Introduction

Text editing is full of small repetitive tasks. Examples include:

- Replace the string “Hashtable” with “Map” throughout a program.
- Reformat a list of phone numbers from “(xxx)yyy-zzzz” to “+1 xxx yyy zzzz”.
- Insert print statements to trace entry and exit from each of a set of functions.
- Generate get/set methods for the instance variables of a class.
- Generate a mailing list from the From headers of a large file of email messages.

Users have a rich basket of tools for automating tasks like these. *Search-and-replace*, in which the user specifies a pattern to search for and replacement text to be substituted, is good enough for simple tasks. *Keyboard macros* are another technique, in which the user records

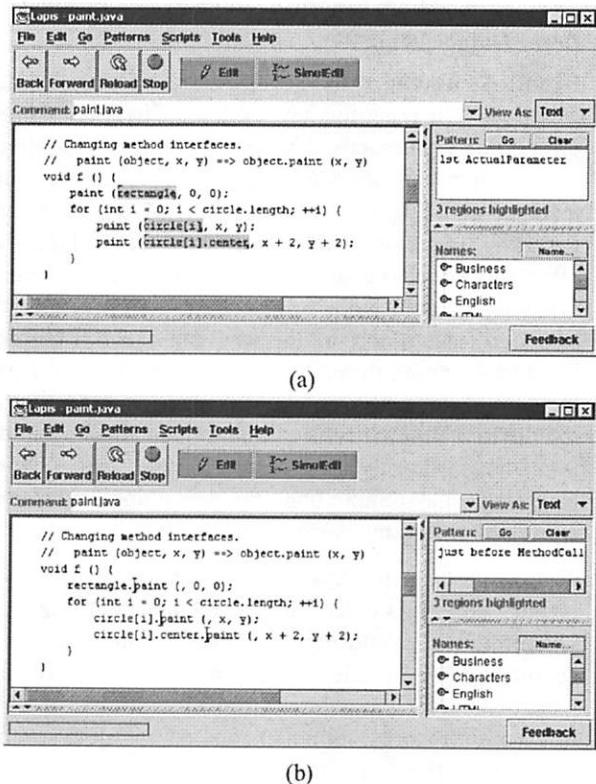


Figure 1: Simultaneous editing on Java code. The records (highlighted lightly) are calls to the `paint()` function, which is being transformed into an object-oriented method. (a) User selects “rectangle”, and the system generalizes the selection across all records. (b) User cuts the selection, pastes it before `paint`, and inserts a dot. The same operation affects every record.

a sequence of keystrokes (or editing commands) and binds the sequence to a single command for easy re-execution. Most keyboard macro systems also support simple loops using tail recursion, where the last step in the macro reinvokes the macro. For more complicated tasks, however, users may resort to a *custom program*, often written in a text-processing language such as Perl, awk, or Emacs Lisp.

This paper proposes a new technique to add to this basket of repetitive text editing tools: *simultaneous editing*. In simultaneous editing, the user first describes a set of regions to edit, called the *records*. This record set can be defined by a pattern, direct selection, or some combination of the two. After defining the records, the user makes a selection in one record using the mouse or keyboard. In response, the system makes an equivalent selection in all other records. Subsequent editing operations – such as typed text, deletions, or cut-and-paste – affect all records simultaneously, as if the user had applied the operations to each record individually. Figure 1 shows simultaneous editing in action.

Simultaneous editing has several advantages over other techniques for repetitive text editing. First, simultaneous editing is interactive. No programming is required. Second, simultaneous editing uses familiar editing commands, including mouse selection. Macro recorders generally ignore or disable mouse selection. Third, the effect of a simultaneous editing operation on any record is readily apparent from the selection. If there is a tricky step in a transformation, the user can check it beforehand by scanning through all records and verifying the location of the selection. Finally, mistakes made in the middle of a simultaneous editing transformation can be immediately detected and corrected with undo. Other techniques may require undoing, debugging, and reexecuting the entire transformation.

The greatest challenge to an implementation of simultaneous editing is determining the equivalent selection where editing should occur in other records. Given a cursor position or selection in one record, the system must generalize it to a description which can be applied to all other records. Simultaneous editing puts several demands on the generalization algorithm:

- Generalization should be fast, so that the system is responsive enough for interactive editing. We solve this problem by preprocessing the records to discover useful features in advance, so that the generalization search for each selection is relatively cheap.
- Generalization should be domain-specific. For example, a user’s selection might best be described in terms of Java syntax. Our solution to this problem is a knowledge base, represented by a library of patterns and parsers that detect structure in text. Users can extend the library on the fly by specifying new patterns, which can be either regular expressions or high-level patterns called *text constraints* [9].
- Generalization should be able to guess accurately from only one example. When multiple general-

izations are consistent with the user’s selection, the generalizer must make its best guess, which hopefully will often be the description the user intended.

- Generalization should be correctable. If the generalizer’s best guess is wrong, the user must have a way to correct it. In our system, the user can select or deselect regions in other records, providing additional positive and negative examples that the generalizer uses to improve its guess. The user can also override the generalizer completely, making a selection by hand or by a pattern.

The rest of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the user interface to simultaneous editing, in the context of an extended example. Section 4 describes some more examples of simultaneous editing. Section 5 delves into the details of our implementation, and Section 6 evaluates its performance. Section 7 outlines some future directions, and Section 8 makes some conclusions.

2 Related Work

Simultaneous editing is similar in concept to Visual Awk [6], a system for developing awk-like file transformers interactively. Like awk, Visual Awk’s default structure consists only of lines and words. When the user selects one or more words in a line, the system highlights the words at the same position in all other lines. For other kinds of selections, the user must select the appropriate tool. For example, Visual Awk’s Cutter tool makes selections by character offset, and its Matcher tool uses regular expressions provided by the user. In contrast, simultaneous editing is built into a conventional text editor, operates on arbitrary records (not just lines), uses standard text editing operations, and automatically infers general, domain-specific descriptions from a user’s selections.

Another closely-related approach to the problem of repetitive text editing is *programming by example*, also called *programming by demonstration* (PBD). In PBD, the user demonstrates one or more examples of the transformation in a text editor, and the system generalizes this demonstration into a program that can be applied to the rest of the examples. PBD systems for text editing have included EBE [12], Tourmaline [11], TELS [13], Eager [1], Cima [7], and DEED [3].

Simultaneous editing is similar to PBD in many ways. Both approaches allow the user to edit with familiar operations, including mouse selection. Both approaches

generalize the user's actions on one example into a description that can be applied to other examples. Both approaches must be able to incorporate multiple examples into the generalization.

However, simultaneous editing has a dramatically different user interface from PBD. In simultaneous editing, the user's demonstration affects all records simultaneously. After demonstrating part of a transformation, the user can scan through the file and see how the other records were affected by the partial transformation. In PBD, on the other hand, each demonstration affects only a single example. In order to see what the inferred program will do to other examples, the user must run the program on other examples. One consequence of this is a lack of trust [1][3]. Users do not trust the inferred program to work correctly on other examples. Although simultaneous editing also does inference, and thus is also susceptible to mistrust, the additional feedback provided by simultaneous selections across all records makes the system's operation more visible, hopefully inspiring more confidence.

The inference used in simultaneous editing is actually *less* powerful than in some PBD systems. TELS, for example, can infer programs containing conditionals and loops. Simultaneous editing assumes just one implicit loop (over the records) and no conditionals (every editing action must be applied to every record). These assumptions permit fast, predictable inference, and allow inference to be applied only to *selections* and not to the sequence of *actions* performed.

Simultaneous editing also requires the user to describe the set of records. The record description is often simple (e.g. lines, or paragraphs, or functions), but some record sets may be hard to describe. By contrast, in most PBD systems, and keyboard macros too, the record set is implicit in the user's demonstration. For example, the demonstration may end with the cursor at the start of the next record.

3 User Interface

This section describes the user interface of simultaneous editing implemented in our prototype system. Features of the user interface will be introduced by presenting an example of the system in operation.

Our implementation of simultaneous editing is built into LAPIS, a text processing system which has been described previously [9][10]. LAPIS has several unusual features that make it well-suited to this effort. First, LAPIS supports multiple simultaneous text selections; most text editors allow only one contiguous selection.

Multiple selections make it easy to display the corresponding selection in every record. Second, LAPIS includes an integrated text pattern language, *text constraints*. Text constraint patterns are convenient not only for the user to describe the record set, but also for the system to describe how it has generalized the user's selection. Finally, LAPIS has a library of built-in parsers and patterns for various kinds of text structure, including HTML and Java source code. The domain knowledge represented by this pattern library enables the system to make its generalizations more accurate and domain-specific, as we will see in the example to follow.

3.1 Example: Get/Set Methods

The example is a common task in Java and C++ programming: for each field *x* of a class (member variable in C++ terminology), create a pair of accessor methods *getX* and *setX* that respectively get and set the value of *x*. Figure 2a shows the original Java class. We want to transform each field declaration so that the variable declaration is followed by its accessor methods, as shown in Figure 2g.

To enter simultaneous editing mode, the user first selects the records to be edited, using multiple selection. A multiple selection can be made two ways in LAPIS: by entering a pattern, which selects all regions that match the pattern; or by holding down the Shift key and selecting text regions with the mouse. In this case, the user chooses *Java.Field* from the pattern library, which runs a Java parser and highlights all field declarations in the current file. If only some of these fields need accessor methods, then the user can either specialize the pattern (e.g. *Java.Field* starting with "public") or manually deselect the undesired fields.

Having selected the records, the user enters simultaneous editing mode by pressing the *SimulEdit* button on the toolbar. The system then does some preprocessing, which involves running all appropriate parsers (such as the Java parser) and searching for interesting features in the selected records. Preprocessing is described in Section 5. The preprocessing delay depends on the number and length of the records. In this simple example, preprocessing takes less than one second. After preprocessing, the editor shows that simultaneous editing is enabled by highlighting the records in yellow.

The user now starts to edit. First, the user clicks at the end of one of the records. The system immediately generalizes this click to the other records, displaying an insertion point at the end of each record (Figure 2b). At the same time, the Pattern box displays a description of the generalization that was made: *point just after Java.Field*. In this case, the description is ac-

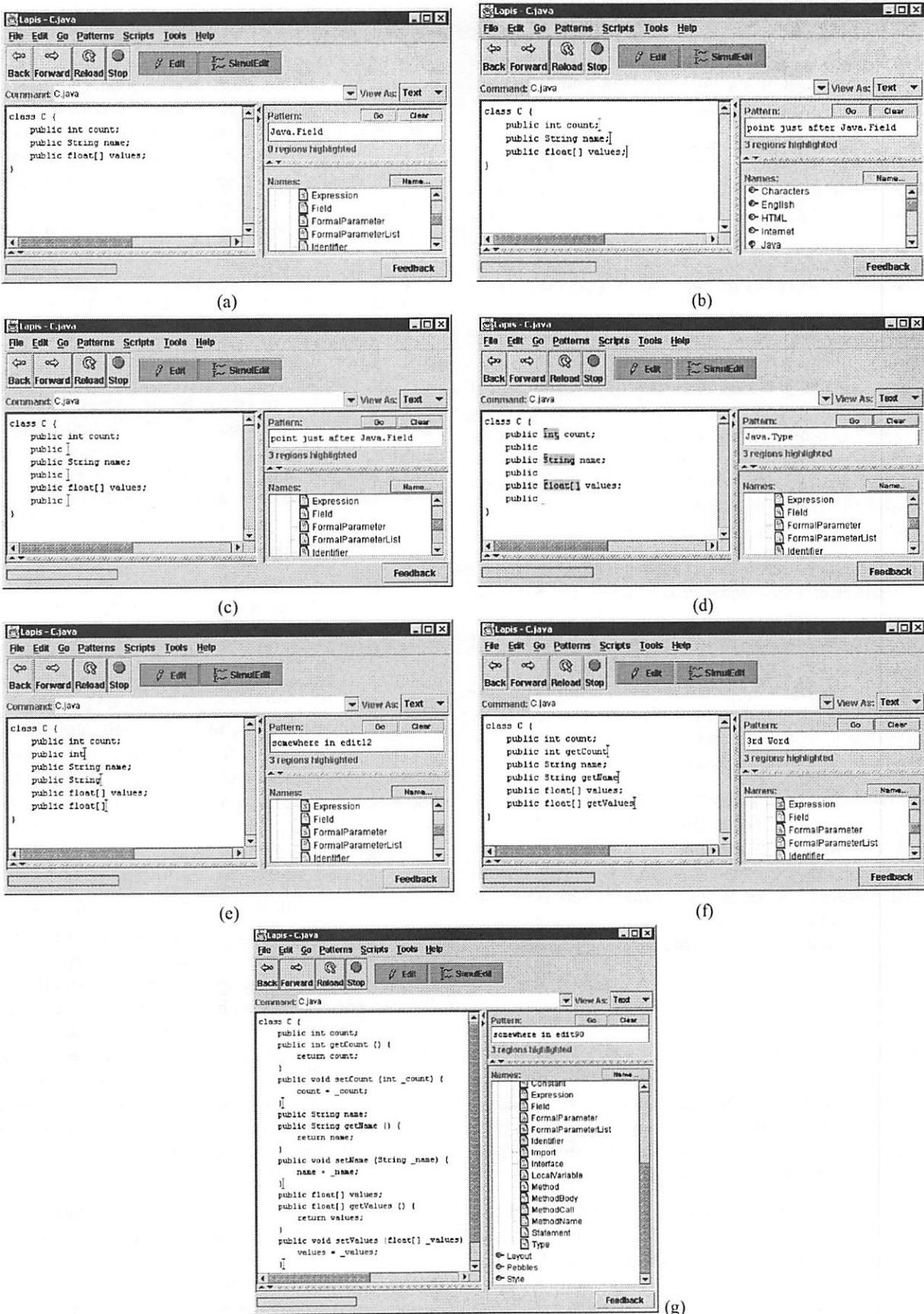


Figure 2: Simultaneous editing used to transform Java field declarations into get/set methods.

tually a text constraint pattern, which could be evaluated to select the same insertion points. The description is not always a valid pattern, because of some design decisions made in our prototype, discussed later. Regardless, the description provides an additional cue for the user to check that the system is properly generalizing the selection.

Having placed the insertion point, the user starts to type in the `getX` method, first pressing Enter to insert a new-line, then indenting a few spaces, then typing “public” to start the method declaration. The typed characters appear in every record (Figure 2c). If typos are made, the user can back up and correct them, using all familiar editing operations. Maintaining the simultaneous insertion points during text entry is trivial, since all records receive the same typed text. No generalization occurs until the user makes a selection somewhere else.

Now the user is ready to enter the return type of the `getX` method. The type is different for each variable `x`, so the user can't simply enter it at the keyboard. Instead, copy-and-paste is used. The user selects the type of one of the fields, in this case, the “int” of “public int `x`”. The system generalizes this selection into the description `Java.Type`, and selects the types of all the other fields (Figure 2d). Note that other generalizations of this selection are possible: “int”, 2nd Word, 2nd from last Word, etc. Some of these generalizations can be discarded immediately because of assumptions of simultaneous editing. For example, “int” does not appear in every record, and so it cannot be selected in every record. Other generalizations are less preferable because they are more complicated than `Java.Type`. In this case, the system's best guess is the right one.

The user then copies the selection to the clipboard, places the insertion point back after “public”, and pastes the clipboard. In response to the copy command, the system copies a *list* of strings to its clipboard, one for each record. When the paste occurs, the system pastes the appropriate string back to each record (Figure 2e).

Similarly, the user copies and pastes the name of variable to create the method name. The lowercase variable `x` is converted into capitalized `X` by applying an editor command that capitalizes the current selection (Figure 2f). Any editor command that applies to a selection or cursor position can be used in simultaneous editing mode.

The rest of `getX` and `setX` are defined by more typing and copy-and-paste commands, until the desired result is achieved (Figure 2g). The user exits simultaneous editing mode by clicking again on the SimulEdit toolbar button, releasing it from the depressed state.

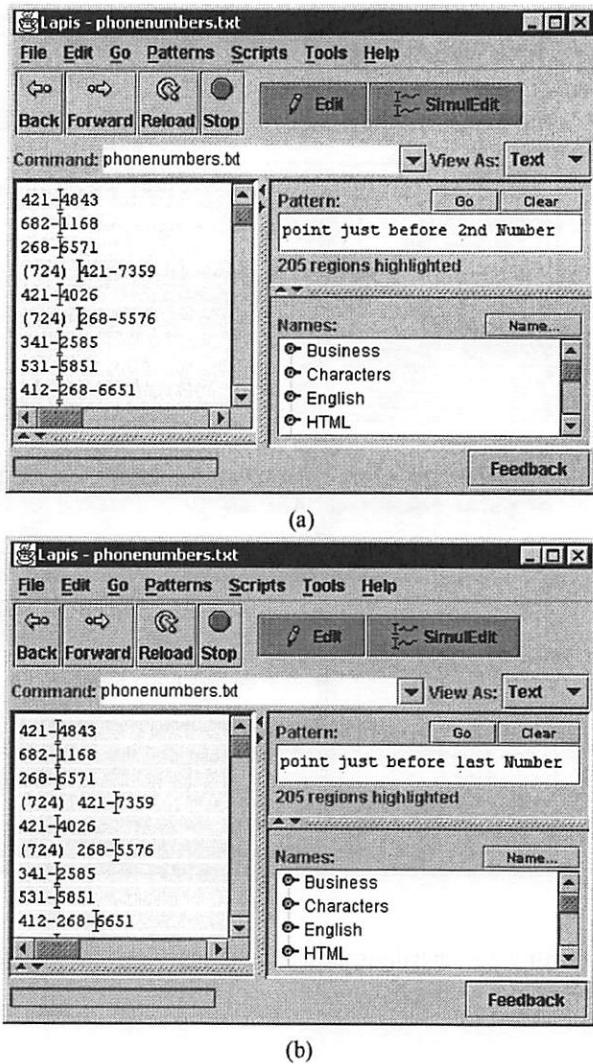


Figure 3: Correcting generalization by switching to a counterexample.

3.2 Correcting Generalizations

The example above raises an important issue: what if the system's generalization was incorrect at some point in the simultaneous editing session? How can the user correct it? Several techniques are available in our system: switching to a counterexample, giving multiple examples, and naming landmarks. These techniques are explained next.

The first correction technique is illustrated in Figure 3. While editing a list of phone numbers, the user tries to place the cursor just before the 4-digit component of each phone number. The first attempt (Figure 3a) is a click before “4843” in the first phone number. This click is incorrectly generalized to `point just before 2nd Number`. An easy way to correct the generalization is to pick one of the records where the gen-

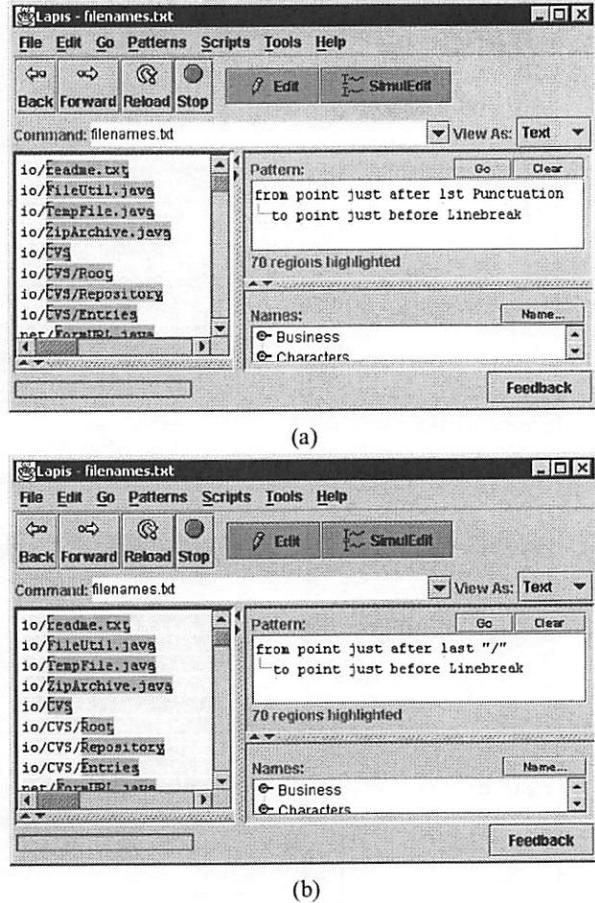


Figure 4: Correcting generalization by providing multiple examples.

eralization failed – for example, a phone number with an area code such as “(724) 421-7359” – and make the selection in that record instead. This selection results in a satisfactory generalization (Figure 3b). This strategy, which we call *switching to a counterexample*, corrects the system by providing a more generic example of the desired selection. The system is still generalizing from only one example; the more generic example replaces the earlier example. An expert user may even avoid the incorrect generalization entirely by selecting the most generic example first.

Sometimes an incorrect generalization cannot be fixed by switching to a counterexample. For example, in Figure 4, the user is trying to select just the filenames, without any directory prefix. Selecting “readme.txt” in the first record generalizes to an incorrect description referring to the point just after 1st Punctuation (Figure 4a). Switching to a counterexample doesn’t work either. For example, selecting “Root” in the sixth record would generalize to last Word, which is also wrong, because it would select only “txt”

in the first record instead of “readme.txt”. To get the desired selection, the user must provide at least two examples of the selection. This is done by holding down the Shift key while selecting the additional example. Alternatively, the user can specify a negative example by deselecting an incorrect selection in another record. Deselecting is done by right-clicking on a selection and picking Unselect from the popup menu that appears. Any number of positive or negative examples can be given. After receiving a new positive or negative example, the system searches for a generalization that selects exactly one region in every record and is consistent with all positive and negative examples. In this case, two positive examples suffice to select the last filename component correctly (Figure 4b).

The user can also assist generalization by making a selection some other way, either by hand or by a pattern, and then assigning it a unique name. The named selection becomes part of the pattern library, where the system can use it as a *landmark* for generalizing other selections. For example, the user might specify a regular expression for the product codes used in his company, and name it ProductCode. Subsequent selections of product codes, or of regions adjacent to product codes, will be generalized much faster and more accurately. This strategy adds more domain knowledge to the system.

Generalization may sometimes fail. There may be insufficient domain knowledge, or the selection may require a more complicated description than the generalizer is designed to generate. For example, our generalizer does not form disjunctions, such as either “gif” or “jpg”. If no generalization can be found that is consistent with the user’s positive and negative examples, then the system gives up, beeps, and leaves only the positive examples selected. No further generalization attempts are made until the user clears the selection and starts a new selection. The user can finish the desired selection by hand, either by selecting the appropriate regions in the other records, or by entering a pattern.

4 Applications

This section describes some applications of simultaneous editing. Two common themes run through these examples. First is the power of *domain knowledge*, such as HTML and Java syntax. Domain knowledge allows the user to specify patterns more concisely and enables the generalizer to make more accurate generalizations with fewer examples. Most text editors either eschew domain knowledge, understanding only low-level concepts like words and characters, or else embed knowledge for only

one domain, such as C++. LAPIIS strives for a middle ground by centralizing domain knowledge into a pattern library that simply generates region sets. Other parts of LAPIIS, such as the generalizer, are domain-independent, and new domain knowledge is easy to add by installing new patterns and parsers in the library.

The second important theme is *interactivity*. Whereas other solutions to these tasks would involve specifying a program and then running it in batch mode, simultaneous editing allows the task to be performed interactively.

4.1 HTML

The following tasks take advantage of the HTML parser included in the pattern library. The HTML parser defines named region sets for each kind of HTML object (e.g. Element, Tag, Attribute) as well as specific HTML tags and attributes (e.g. ``, `href`).

Change all `` elements into `` elements. The user runs the pattern `Bold` to select all bold elements (which look like `bold text`), then enters simultaneous editing mode. The user selects the first `b` with the mouse (which the system generalizes to "`b`" in "``"), deletes it, and types in "`strong`". The user then selects the last `b` (which generalizes to "`b`" in "``"), deletes it, and types in "`strong`" again.

Convert HTML to XHTML. One difference between HTML and XHTML (an XML format) is the treatment of tags with no content, such as ``, `
`, or `<hr>`. In XHTML, elements with no end tag should be written as `` so that an XML parser can parse them without access to the XHTML document type definition. Making this conversion with simultaneous editing is straightforward. To select the empty tags, the user runs the pattern `Tag = Element`, which matches all regions that the HTML parser identified as both tags and complete elements. Entering simultaneous editing mode, the user places the cursor at the end of the tag (which generalizes to `point just before ">"`) and inserts a slash to finish the task.

4.2 Source Code

Programming is full of tasks where simultaneous editing is useful, particularly when given knowledge of the language syntax. The examples below are in Java because LAPIIS has a Java parser in its library. Other languages could be edited in similar fashion by adding an appropriate parser to the library.

Change access permissions. Suppose a class contains a number of fields or methods that currently have default access permission, and the programmer wants

to change their permissions to `private`. The programmer selects the relevant fields and methods (using, for instance, (Field or Method) not starting with "public" or "private"), enters simultaneous editing mode, and types `private` at the beginning of a field, changing all the others simultaneously.

Change a method interface. If a method's parameters change, then simultaneous editing can be used to rewrite all the calls to that method at once. For example, suppose a method `copy(src, dest)` must be changed to `copy(dest, src)` for consistency with similar interfaces in the program. The programmer selects all calls to `copy` (perhaps using the pattern `MethodCall` starting with `Identifier` equal to "`copy`"), enters simultaneous editing mode, selects the first argument (which generalizes to `first ActualParameter`), copies it to the clipboard, and then pastes it after the second argument. A little more editing fixes the comma separators, and the change is done. This example demonstrates how simultaneous editing with domain knowledge can deliver the power of syntax-directed editing inside a freeform text editor.

Wrap every method with entry and exit code. While debugging a class, the programmer wants to run some code whenever any method of the class is entered or exited. This code might do tracing (printing the method name to a log), performance timing, or validation (checking that the method preserves class invariants). To add this code, the programmer selects all the methods using the pattern `Method` and starts simultaneous editing. Next, the programmer types in the entry code at the start of the method, wraps the rest of the method body with a `try-finally` construct, and types the exit code inside the `finally` clause. All the methods change identically. This kind of modification is an example of *aspect-oriented programming* [5], where code is "woven" into a program at program locations described by a pattern.

4.3 Scripting

To understand the next set of examples, the reader should be aware that LAPIIS is also a shell [10]. An external command can be executed in the LAPIIS command box, drawing its standard input from the current contents of the editor and sending its output back to the editor.

Disposable scripts. Suppose the user wants to make a group of GIF files transparent using `giftrans`. Simultaneous editing offers a solution based on the idea of creating a one-time script directly from data. The user first runs `ls *.gif` to list the relevant filenames in the

editor. Using simultaneous editing, the user edits each line into a command, such as `giftrans -T X.gif > X-transparent.gif`. Then the user runs the resulting script with `sh`. Disposable scripts are a more interactive way to achieve the effect of the Unix commands `foreach` or `xargs`.

Impedance matching. Data obtained from the output of one program must often be massaged before it can be fed into another program. Simultaneous editing offers the opportunity to perform this massaging interactively, which is particularly sensible for one-shot tasks. For example, suppose a user is testing network connectivity with `traceroute`, and wants to pass the network latencies computed by `traceroute` into `gnuplot` to generate a graph. The user first runs `traceroute` to generate a trace. Using simultaneous editing, the user edits each line of the trace, leaving only the hop number (1, 2, ...) and the latency time. After exiting simultaneous editing mode, the user inserts a `gnuplot` plot instruction before the first line (`plot "-" with lines`) and finally runs `gnuplot -persist` to plot the data.

5 Implementation

This section describes the algorithm used to generalize the user's selection to a description that can be applied to all records. The input to the generalizer is a set of positive examples, a set of negative examples, and the set of records. The output is a selection consistent with the positive and negative examples that selects exactly one region in every record, plus a human-readable description of the selection.

Like other PBD systems, the generalizer basically searches through a space of hypotheses for a hypothesis consistent with the examples. The details of the implementation are novel, however. Our generalizer is actually split into three parts: preprocessing, search, and updating. Preprocessing takes the set of records as input and generates a list of useful features as output. Preprocessing takes place only once, when the user first enters simultaneous editing mode. The search phase takes the positive and negative examples and the feature list generated by preprocessing, and computes a selection consistent with the examples. Search happens whenever the user makes a selection with the mouse or keyboard, or adds a new positive or negative example to the current selection. Finally, updating occurs when the user edits the records by inserting, deleting, or copying and pasting text. Updating takes the user's edit action as input and modifies the feature list appropriately. Each of these phases is described in more detail below.

5.1 Region Sets

Before describing the generalizer, we first briefly describe the representations used for selections in a text file. More detail can be found in an earlier paper about LAPIS [9]. A *region* $[s, e]$ is a substring of a text file, described by its start offset s and end offset e relative to the start of the text file. A *region set* is a set of regions.

LAPIS has two novel representations for region sets. First, a *fuzzy region* is a four-tuple $[s_1, s_2; e_1, e_2]$ that represents the set of all regions $[s, e]$ such that $s_1 \leq s \leq s_2$ and $e_1 \leq e \leq e_2$. Note that any region $[s, e]$ can be represented as the fuzzy region $[s, s; e, e]$. Fuzzy regions are particularly useful for representing relations between regions. For example, the set of all regions that are inside $[s, e]$ can be compactly represented by the fuzzy region $[s, e; s, e]$. Similar fuzzy region representations exist for other relations, including *contains*, *before*, *after*, *just before*, *just after*, *starting* (i.e. having coincident start points), and *ending*. These relations are fundamental operators in the text constraints pattern language, and are also used in generalization.

The second novel representation is the *region tree*, a union of fuzzy regions stored in an R-tree in lexicographic order [9]. A region tree can represent an arbitrary set of regions, even if the regions nest or overlap each other. A region tree containing N fuzzy regions takes $O(N)$ space, $O(N \log N)$ time to build, and $O(\log N)$ time to test a region for membership in the set.

These two representations are used by the preprocessing phase to construct a list of features that the search phase can use to quickly test positive and negative examples. The selection returned by the search phase is also represented as a region set.

5.2 Feature Generation

Preprocessing takes the set of records and generates a list of useful features. A feature is a region set, containing at least one region in each record, where the regions are in some sense equivalent. For example, the feature `Java . Type` represents the set of all regions that were recognized by the Java parser as type names. The preprocessor generates two kinds of features: *pattern features* derived from the pattern library, and *literal features* discovered by examining the text of the records.

Pattern features are found by applying every parser and every named pattern in the pattern library. LAPIS has a considerable library of built-in parsers and patterns, including Java, HTML, character classes (e.g. digits, punctuation, letters), English structure (words, sentences, paragraphs), and various codes (e.g., URLs,

email addresses, hostnames, IP addresses, phone numbers). The user can readily add new named patterns and new parsers. The result of applying a pattern is the set of all regions matching the pattern. The result of applying a parser is a collection of named region sets. For example, the Java parser generates region sets for Statement, Expression, Type, Method, and so on.

After applying all library patterns, the preprocessor discards any patterns that do not have at least one match in every record. This is justified by two assumptions made by the generalizer: first, that a generalization must have at least one match in every record; and second, that a generalization can be represented without disjunction. Given these two assumptions, only features that match somewhere in every record will be useful for constructing generalizations.

By the same reasoning, the only useful literal features are common substrings, i.e. substrings that occur at least once in every record. Common substrings can be found efficiently using a *suffix tree* [4]. A suffix tree is a path-compressed trie into which all suffixes of a string have been inserted. With a suffix tree for a string s , we can test whether a substring p occurs in s in only $O(|p|)$ time. Naive suffix tree construction (inserting every suffix explicitly) takes $O(|s|^2)$ time, which is sufficient for our prototype since records tend to be short. Several algorithms exist for building a suffix tree in linear time, however [4], and extending the algorithm below to accommodate them would be straightforward.

The common substring algorithm works as follows. The algorithm starts by building a suffix tree from the shortest record, in order to minimize the size of the initial suffix tree. This suffix tree represents the set of common substrings of all records examined so far. For each of the remaining records, the suffixes of the record are matched against the suffix tree one by one. Each tree node keeps a count of the number of times it was visited during the processing of the current record. This count represents the number of occurrences (possibly overlapping) of the substring represented by the node. After processing each record, all unvisited nodes are pruned from the tree, since the corresponding substrings never occurred in the record. After processing every record in this fashion, the only substrings left in the suffix tree must be common to all the records. These common substrings are used as literal features. The operation of the common substring algorithm is illustrated in Figure 5.

5.3 Feature Ordering

After generating useful features from the set of records, the preprocessor sorts them into a list in order of preference. Placing the most-preferred features first makes

the search phase simpler. The search can just scan down the list of features and stop as soon as it finds the first feature consistent with the examples, since this feature is guaranteed to be the most preferred consistent feature.

Features are classified into three groups for the purpose of preference ordering: *unique features*, which occur exactly once in each record; *regular features*, which occur exactly n times in each record, for some $n > 1$; and *varying features*, which occur a varying number of times in each record. A feature's classification is not predetermined. Instead, it is found by actually counting occurrences in the records being edited. For example, in Figure 2, `Java.Type` is a unique feature, since it occurs exactly once in every variable declaration. Regular features are commonly found as delimiters. For example, if the records are IP addresses like `127.0.0.1`, then `"` is a regular feature. Varying features are typically tokens, like words and numbers, which are general enough to occur in every record but do not necessarily follow any regular pattern.

Unique features are preferred over the other two kinds. A unique feature has the simplest description: the feature name itself, such as `Java.Type`. By contrast, using a regular or varying feature in a generalization requires specifying the index of a particular occurrence, such as `5th Word`. Regular features are preferred over varying features, because regularity of occurrence is a strong indication that the feature is relevant to the internal structure of a record.

Within each group, pattern features are preferred over literal features. We also plan to let the user specify preferences between pattern features, so that, for instance, Java features can be preferred over character-class features. We are still designing the user interface for this, however, so the prototype currently leaves pattern features in an arbitrary order. Among literal features, longer literals are preferred to shorter ones.

To summarize, the preprocessor orders the feature list in the following order, with most preferred features listed first: unique patterns, unique literals, regular patterns, regular literals, varying patterns, varying literals. Within each group of patterns, the order is arbitrary. Within each group of literals, longer literals are preferred to shorter.

5.4 Search

The search algorithm takes the user's positive and negative examples and the feature list generated by preprocessing, and attempts to generate a description consistent with the examples.

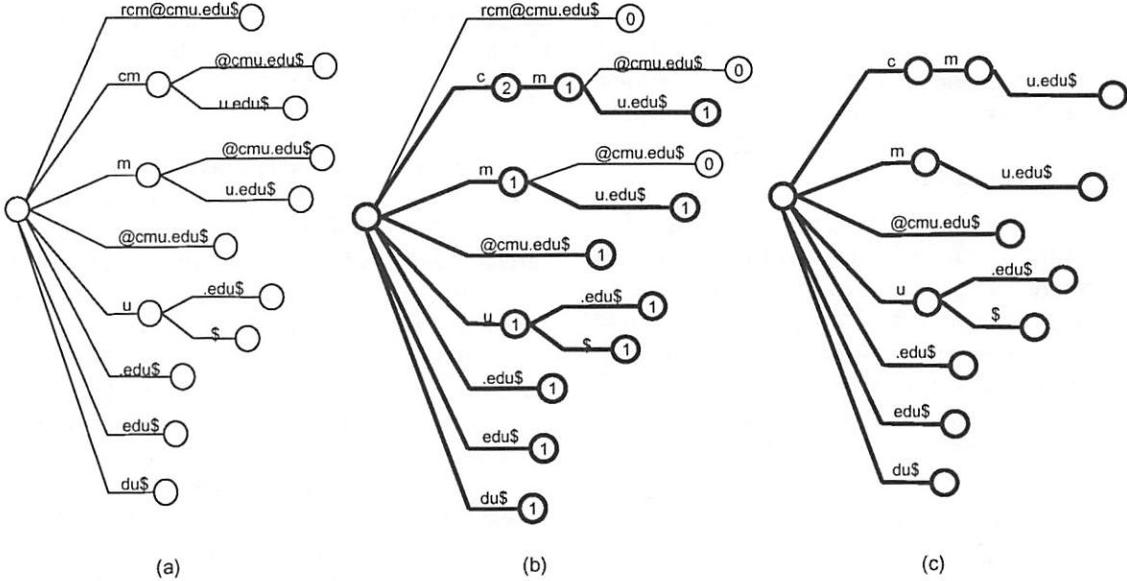


Figure 5: Finding common substrings using a suffix tree. (a) Suffix tree constructed from first record, `rcm@cmu.edu`; \$ represents a unique end-of-string character. (b) Suffix tree after matching against second record, `ljc@cmu.edu`. Each node is labeled by its visit count. (c) Suffix tree after pruning nodes which are not found in `ljc@cmu.edu`. The remaining nodes represent the common substrings of the two records.

The basic search process works as follows. The system chooses the first positive example, called the *seed example*, and scans through the feature list, testing the seed example for membership in each feature. Since each feature is represented by a region tree, this membership test is very fast. When a matching feature is found, the system constructs one or more candidate descriptions representing the particular occurrence that matched. For example, if the seed example matches the (varying) feature `Word`, the system might construct the candidate descriptions `5th Word` and `2nd from last Word` by counting words in the seed example's record. These candidate descriptions are tested against the other positive and negative examples, if any. The first consistent description found is returned as the generalization.

The output of the search process depends on whether the user is selecting an insertion point (e.g. by clicking the mouse) or selecting a region (e.g. by dragging). If all the positive examples are zero-length regions, then the system assumes that the user is placing an insertion point, and searches for a point description. Otherwise, the system searches for a region description.

To search for a point description, the system transforms the seed example, which is just a character offset b , into two fuzzy regions: $[b, b; b, +\infty]$, which represents all regions that start at b , and $[-\infty, b; b, b]$, which represents all regions that end at b . The search tests these fuzzy

regions for intersection with each feature in the feature list, which is just as fast as a simple region membership test. Candidate descriptions generated by the search are transformed into point descriptions by prefixing point just before or point just after, depending on which fuzzy region matched the feature, and then the descriptions are tested for consistency with the other positive and negative examples.

To search for a region description, the system first searches for the seed example using the basic search process described above. If no matching feature is found – because the seed example does not correspond precisely to a feature on the feature list – then the system splits the seed example into its start point and end point, and recursively searches for point descriptions for each point. Candidate descriptions for the start point and end point are transformed into a description of the entire region by wrapping with `from...to...`, and then tested for consistency with the other examples.

This search algorithm is capable of generalizing a selection only if it starts and ends on a feature boundary. For literal features, this is not constraining at all. Since a literal feature is a string that occurs in all records, every substring of a literal feature is also a literal feature. Thus every position in a literal feature lies on a feature boundary. To save space, the preprocessor only stores maximal literal features in the feature list, and the search phase

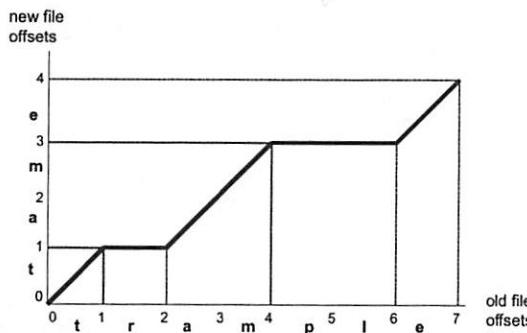


Figure 6: Coordinate map translating offsets between two versions of a file. The old version is the word `trample`. Two regions are deleted to get the new version, `tame`.

tests whether the seed example falls anywhere inside a maximal literal feature.

5.5 Updating

In simultaneous editing, the user is not only making selections, but also editing the file. Editing has two effects on generalization. First, every edit changes the start and end offsets of regions. As a result, the region sets used to represent features become invalid. Second, editing changes the file content, so the precomputed features may become incomplete or wrong. For example, if the user types some new words, then the precomputed Word feature becomes incomplete, since it doesn't include the new words the user typed. The updating algorithm addresses these two problems.

From the locations and length of text inserted or deleted, the updating algorithm computes a *coordinate map*, a relation that translates a file offset prior to the change into the equivalent file offset after the change. The coordinate map can translate coordinates in either direction. For example, Figure 6 shows the coordinate map for a simple edit. Offset 3 in `trample` corresponds to offset 2 in `tame`, and vice versa. Offsets with more than one possible mapping in the other version, such as offset 1 in `tame`, are resolved arbitrarily. Our prototype picks the largest value.

Since the coordinate map for a group of insertions or deletions is always piecewise linear, it can be represented as a sorted list of the (x,y) endpoints of each line segment. If a single edit consists of m insertions or deletions (one for each record), then this representation takes $O(m)$ space. Evaluating the coordinate map function for a single offset takes $O(\log m)$ time using binary search.

A straightforward way to use the coordinate map is to scan down the feature list and update the start and end

points of every feature to reflect the change. If the feature list is long, however, and some feature sets are large (such as Word or Whitespace), the cost of updating every feature after every edit may be prohibitive. Our generalizer takes the opposite strategy: instead of translating all features up to the present, we translate the user's positive and negative examples *back to the past*. The system maintains a global coordinate map representing the translation between original file coordinates (when simultaneous editing mode was entered and the feature list generated) and the current file coordinates. When an edit occurs, the updating algorithm computes a coordinate map for the edit and composes it with this global coordinate map. When the user provides positive and negative examples to generalize into a selection, the examples are translated back to the original file coordinates using the inverse of the global coordinate map. The search algorithm generates a consistent description for the translated examples. The generated description is then translated forward to current file coordinates before being displayed as a selection.

An important design decision in a simultaneous editing system that uses domain knowledge, such as Java syntax, is whether the system should attempt to reparse the file while the user is editing it. On one hand, reparsing allows the generalizer to track all the user's changes and reflect those changes in its descriptions. On the other hand, reparsing is expensive and may fail if the file is in an intermediate, syntactically incorrect state. Our generalizer never reparses automatically in simultaneous editing mode. The user can explicitly request reparsing with a menu command, which effectively restarts simultaneous editing using the same set of records. Otherwise, the feature list remains frozen in the original version of the file. One consequence of this decision is that the generalizer's human-readable descriptions may be misleading because they refer to an earlier version.

This design decision raises an important question. If the feature list is frozen, how can the user make selections in newly-inserted text, which didn't exist when the feature list was built? This problem is handled by the update algorithm. Every typed insert in simultaneous editing mode adds a new literal feature to the feature list, since the typed characters are guaranteed to be identical in all the records. Similarly, pasting text from the clipboard creates a special feature that translates coordinates back to the source of the paste and tries to find a description there. When the generalizer uses one of these features created by editing, the feature is described as "somewhere in editN", which can be seen in Figures 2e and 2g.

A disadvantage of this scheme is that the housekeeping structures – the global coordinate map and the new

features added for edits – grow steadily as the user edits. This growth can be slowed significantly by coalescing adjacent insertions and deletions, although we have not yet implemented this. Another solution might be to reparse when the number of edits reaches some threshold, doing the reparsing in the background on a copy of the file in order to avoid interfering with the user’s editing. In practice, however, we don’t expect space growth to be a serious problem. In all the applications we have imagined, the user spends only a few minutes in a simultaneous editing session, not the hours that are typical of general text editing. After leaving simultaneous editing mode, the global coordinate map and the feature list can be discarded.

6 Evaluation

Simultaneous editing was evaluated by a small user study. Eight users were found by soliciting campus newsgroups. All were college undergraduates with substantial text-editing experience and varying levels of programming experience (5 described their programming experience as “little” or “none,” and 3 as “some” or “lots”). Users were paid for participation. Users first learned about simultaneous editing by reading a tutorial and trying the examples. The tutorial took less than 10 minutes for all but one user (who spent 30 minutes exploring the system). After the tutorial, each user performed the following three tasks:

1. Put the author name and publication year in front of each citation.

Before:

1. Aha, D.W. and Kibler, D. Noise-tolerant instance-based learning algorithms. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 1989, pp. 794-799
 2. Hayes-Roth, B., Pfleger, K., Morignot, P., and Lalanda, P. Plans and Behavior in Intelligent Agents. Technical Report KSL-95-35, Stanford University, 1995.
 ... (7 more) ...

After:

[Aha 89] Aha, D.W. and Kibler, D. Noise-tolerant instance-based learning algorithms. In Proceedings of the Eleventh International Joint Conference on Artificial Intelligence. Morgan Kaufmann, 1989, pp. 794-799.
 [Hayes-Roth 95] Hayes-Roth, B., Pfleger, K., Morignot, P., and Lalanda, P. Plans and Behavior in Intelligent Agents, Technical Report KSL-95-35, Stanford University, 1995.
 ... (7 more) ...

2. Reformat a list of mail aliases from HTML to text.

Before:

<DT>
 Conceptual Graphs
 <DT>
 KIF
 ... (5 more) ...

After:

:: Conceptual Graphs
 congra: mailto:cg@cs.umn.edu
 :: KIF
 kif: mailto:kif@cs.stanford.edu
 ... (5 more) ...

3. Reformat a list of baseball scores into a tagged format (7 records).

Before:

Cardinals 5, Pirates 2.
 Red Sox 12, Orioles 4.
 ... (5 more) ...

After:

GameScore[winner ‘Cardinals’; loser ‘Pirates’; scores[5, 2]].
 GameScore[winner ‘Red Sox’; loser ‘Orioles’; scores[12,4]].
 ... (5 more) ...

All tasks were obtained from other authors (tasks 1 and 2 from Fujishima [3] and task 3 from Nix[12]). After performing a task with simultaneous editing, users repeated the task with manual editing, but only on the first three records to avoid unnecessary tedium. Users were instructed to work carefully and accurately at their own pace. All users were satisfied that they had completed all tasks, although the finished product sometimes contained undetected errors, a problem discussed further below. No performance differences were seen between programmers and nonprogrammers. Aggregate times for each task are shown in Table 1.

Following the analysis used by Fujishima [3], we estimate the leverage obtained with simultaneous editing by dividing the time to edit all records with simultaneous editing by the time to edit just one record manually. This ratio, which we call *equivalent task size*, represents the number of records for which simultaneous editing time would be equal to manual editing time for a given user. Since manual editing time increases linearly with record number and simultaneous editing time is roughly constant (or only slowly increasing), simultaneous editing will be faster whenever the number of records is greater than the equivalent task size. (Note that the average equivalent task size is not necessarily equal to the ratio of the average editing times, since $E[S/M] \neq E[S]/E[M]$.)

As Table 1 shows, the average equivalent task sizes are small. In other words, the average novice user works faster with simultaneous editing if there are more than 8.4 records in the first task, more than 3.6 records in the second task, or more than 4 records in the third task.¹ Thus simultaneous editing is an improvement over manual editing even for very small repetitive editing tasks, and even for users with as little as 10 minutes of experience. Some users were so slow at manual editing that their equivalent task size is smaller than the expert’s, so simultaneous editing benefits them even more. Simultaneous editing also compares favorably to another PBD system, DEED [3]. When DEED was evaluated with

¹These estimates are actually conservative. Simultaneous editing always preceded manual editing for each task, so the measured time for simultaneous editing includes time spent thinking about and understanding the task. For the manual editing part, users had already learned the task, and were able to edit at full speed.

Task	Records in task	Simultaneous editing	Manual editing	Equivalent task size	
				novices	expert
1	9	142.9 s [63-236 s]	21.6 s/rec [7.7-65 s/rec]	8.4 recs [2.1-12.2 recs]	4.5 recs
2	7	119.1 s [64-209 s]	32.3 s/rec [19-40 s/rec]	3.6 recs [1.9-5.8 recs]	1.6 recs
3	7	159.6 s [84-370 s]	41.3 s/rec [16-77 s/rec]	4.0 recs [1.9-6.2 recs]	2.4 recs

Table 1: Time taken by users to perform each task (mean [min-max]). *Simultaneous editing* is the time to do the entire task with simultaneous editing. *Manual editing* is the time to edit 3 records of the task by hand, divided by 3 to get a per-record estimate. *Equivalent task size* is the ratio between simultaneous editing time and manual editing time for each user; *novices* are users in the user study, and *expert* is one of the authors, provided for comparison. A task with more records than *equivalent task size* would be faster with simultaneous editing than manual editing.

novice users on tasks 1 and 2, the reported equivalent task sizes averaged 42 and ranged from 5 to 200, which is worse on average and considerably more variable than simultaneous editing.

Another important part of system performance is generalization accuracy. Each incorrect generalization forces the user to make at least one additional action, such as selecting a counterexample or providing an additional positive or negative example. In the user study, users made a total of 188 selections that were used for editing. Of these, 158 selections (84%) were correct immediately, requiring no further examples. The remaining selections needed either 1 or 2 extra examples to generalize correctly. On average, only 0.26 additional examples were needed per selection. Unfortunately, users tended to overlook slightly-incorrect generalizations, particularly generalizations that selected only half of the hyphenated author “Hayes-Roth” or the two-word baseball team “Red Sox”. As a result, the overall error rate for simultaneous editing was slightly worse than for manual editing: 8 of the 24 simultaneous editing sessions ended with at least one uncorrected error, whereas 5 of 24 manual editing sessions ended with uncorrected errors. If the two most common selection errors had been noticed by users, the error rate for simultaneous editing would have dropped to only 2 of 24. We are currently studying ways to call the user’s attention to possible selection errors [8].

After doing the tasks, users were asked to evaluate the system’s ease-of-use, trustworthiness, and usefulness on a 5-point Likert scale. These questions were also borrowed from Fujishima [3]. The results, shown in Figure 7, are generally positive.

7 Status and Future Work

Simultaneous editing has been implemented in LAPIPS, a browser/editor designed for processing structured text. LAPIPS is written in Java 1.1, extending the JFC text editor component JEditorPane. Directions for obtaining LAPIPS are found at the end of this paper.

We have many ideas for future work. First and perhaps most challenging is the problem of scaling up to large tasks. Although our prototype is far from a toy, since it can handle 100KB files with relative ease, many interesting tasks involve megabytes of data spread across multiple files. Large data sets pose several problems for simultaneous editing. The first problem is system responsiveness. Making a million edits with every keystroke may slow the system down to a crawl, particularly if the text editor uses a *gap buffer* to store the text [2]. Gap buffers are used by many editors, among them Emacs and JEditorPane, the Java class on which our prototype is based. With a gap buffer and a record set that spans the entire file, typing a single character forces the editor to move nearly every byte in the buffer. One way to address this problem is to delay edits to the rest of the file until the user scrolls. Another solution would be to have multiple gaps in the buffer, one for each record.

Another problem with large files is checking for incorrect generalizations. When editing a small file, the user can just scan through the entire file to ensure that a selection has been generalized properly. With a large file, scanning becomes infeasible. We have several ideas for secondary visualizations that might help with this problem. One is a “bird’s-eye view” showing the entire file (in greeked text), so that deviations in an otherwise regular highlight can be noticed at a glance. Another is an abbreviated context view, showing only the selected lines from each record. A third view is an “unusual matches” view, showing only the most unusual examples of the generalization, found by clustering the matches [8].

A third problem with large data sets is where the data resides. For interactive simultaneous editing, the data must fit in RAM, with some additional overhead for parsing and storing feature lists. For large data sets, this is impractical. However, it is easy to imagine interactively editing a small sample of the data to record a macro which is applied in batch mode to the rest of the data. The batch mode could minimize its memory requirements by reading and processing one record at a time (or one translation unit at a time, if it depends on a Java

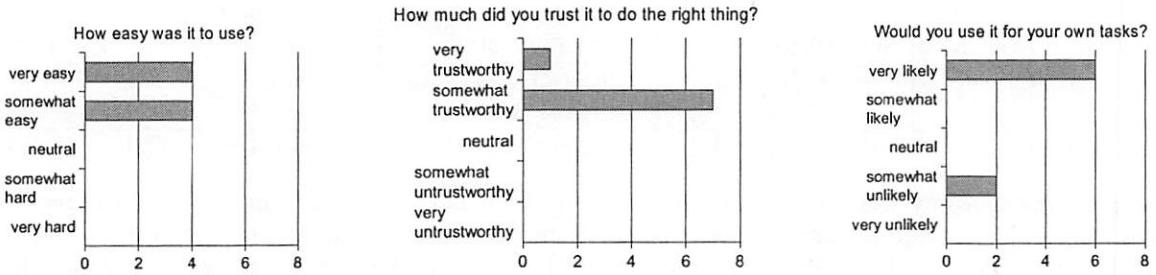


Figure 7: User responses to questions about simultaneous editing.

or HTML parser). Macros recorded from simultaneous editing would most likely be more reliable than keyboard macros recorded from single-cursor editing, since simultaneous editing finds general patterns representing each selection. The larger and more representative the sample used to demonstrate the macro, the more correct the patterns would be. The macro could also be saved for later reuse.

8 Conclusions

Simultaneous editing is an effective way for users to perform repetitive text editing tasks interactively, using familiar editing commands. Its combination of interactivity and domain specificity makes simultaneous editing a useful addition to our basket of tools for text processing, which is practical for inclusion in a wide variety of editors.

The LAPIS browser/editor, which includes an implementation of simultaneous editing with Java source code, may be downloaded from

<http://www.cs.cmu.edu/~rcm/lapis/>

Acknowledgements

The authors are indebted to Yuzo Fujishima for providing the materials to reproduce the DEED user study. We would also like to thank Laura Cassenti, Sarit Sotangkur, Dorothy Zaborowski, Brice Cassenti, and Jean Cassenti for enduring early versions of simultaneous editing, and Sheila Harnett and the anonymous referees for their helpful comments. This research was funded in part by USENIX Student Research Grants.

References

- [1] A. Cypher. Eager: Programming repetitive tasks by demonstration. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 205–218. MIT Press, 1993.
- [2] C. A. Finseth. Theory and practice of text editors, or, a cookbook for an EMACS. Technical Memo 165, MIT Lab for Computer Science, May 1980.
- [3] Y. Fujishima. Demonstrational automation of text editing tasks involving multiple focus points and conversions. In *Proceedings of Intelligent User Interfaces '98*, pages 101–108, 1998.
- [4] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.
- [6] J. Landauer and M. Hirakawa. Visual AWK: a model for text processing by demonstration. In *Proceedings of the 11th International IEEE Symposium on Visual Languages '95*, September 1995.
- [7] D. Maulsby. *Instructible Agents*. PhD thesis, University of Calgary, 1994.
- [8] R. C. Miller. *Lightweight Structured Text Processing*. PhD thesis, Carnegie Mellon University, 2001. In preparation.
- [9] R. C. Miller and B. A. Myers. Lightweight structured text processing. In *USENIX 1999 Annual Technical Conference*, pages 131–144, June 1999.
- [10] R. C. Miller and B. A. Myers. Integrating a command shell into a web browser. In *USENIX 2000 Annual Technical Conference*, pages 171–182, June 2000.
- [11] B. A. Myers. Tourmaline: Text formatting by demonstration. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 309–322. MIT Press, 1993.
- [12] R. Nix. Editing by example. *ACM Transactions on Programming Languages and Systems*, 7(4):600–621, October 1985.
- [13] I. H. Witten and D. Mo. TELS: Learning text editing tasks from examples. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 183–204. MIT Press, 1993.

High-Performance Memory-Based Web Servers: Kernel and User-Space Performance

Philippe Joubert*, Robert B. King†, Rich Neves*, Mark Russinovich‡, John M. Tracey§

*IBM, T. J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598*

Abstract

Web server performance has steadily improved since the inception of the World Wide Web. We observe performance gains of two orders of magnitude between the original process-based Web servers and today's threaded Web servers. Commercial and academic Web servers achieved much of these gains using new or improved event-notification mechanisms and techniques to eliminate reading and copying data, both of which required new operating system primitives. More recently, experimental and production Web servers began integrating HTTP processing in the TCP/IP stack and providing zero copy access to a kernel-managed cache. These kernel-mode Web servers improved upon newer user-mode Web servers by a factor of two to six.

This paper analyzes the significant performance gap between the newer user-mode and kernel-mode Web servers on Linux and Windows 2000. Several user-mode and kernel-mode Web servers are compared in three areas: data movement, event notification, and communication code path. To establish a user-mode baseline, the paper measures the performance of highly optimized Web servers. The paper positions these user-mode implementations with those from related research projects. In particular, the "Adaptive Fast Path Architecture" (AFPA) is described and then used to implement kernel-mode Web servers on Linux and Windows 2000. AFPA is a platform for implementing kernel-mode network servers on production operating systems without kernel modifications. AFPA runs on Linux, Windows 2000, AIX, and S/390. The results show that kernel-mode performance greatly

exceeds the performance of user-mode servers implementing a variety of performance optimizations. The paper concludes that significant opportunities remain to bridge the gap between user-mode and kernel-mode Web server performance.

1 Introduction

Increasing demand for Web content and services has motivated techniques to grow Web server capacity. As a result, Web server performance has steadily improved and Web-hosting infrastructure has become more complex. Today's Web "farms" are multi-tiered and employ several types of specialized server systems dedicated to caching static content, applications, and databases. For example, network service providers use proxy caches to geographically distribute content on behalf of specific customers. This reduces bandwidth costs and improves end-user response times. Akamai [1], for instance, has built a commercial service for caching portions of their customer's static content using geographically distributed edge caches. In addition to caching static content, Web servers are able to cache dynamic content such as price lists, stock quotes, or sports scores. Dynamic content often changes at a coarse enough frequency or requires publishing at intervals sufficient for caching. Commercial efforts [2] and research projects [3] have successfully exploited the ability to cache most forms of dynamic content at the front tier of a Web delivery architecture. While some forms of dynamic content have real-time publishing requirements and remain difficult to cache, it has been shown by [4] that the ratio between all forms of dynamic and static content has remained constant, defying the commonly-held belief that dynamic workloads will dominate over time.

Caching Web servers are well-suited to analyzing network server performance tradeoffs. These servers are

*Philippe Joubert and Rich Neves' current affiliation is ReefEdge Inc. email: philippe.rich@reefedge.com

†email: rbking2@us.ibm.com

‡Mark Russinovich's current affiliation is Winternals Software, 3101 Bee Caves Rd, Austin TX 78746. email: mark@sysinternals.com

§email: traceyj@us.ibm.com

simple to implement and measure with existing, unmodified static Web benchmarks. Caching Web servers reduce network server logic to parsing an HTTP GET request. It's possible to parse such a request with a few lines of C code. What remains is experimentation with thread models, scheduling mechanisms, and new operating system primitives to reduce memory copies. The simplicity of parsing HTTP GET requests reduces the complexity of the code required for kernel-mode caching Web servers. Furthermore, it's usually possible to implement kernel-mode caching Web servers without modifying the operating system kernel, providing useful control cases for assessing user-mode optimizations.

This paper analyzes the performance gap between the fastest currently available user-mode caching Web servers and their kernel-mode counterparts while holding the operating system and hardware fixed. The goal of this analysis is to identify the potential performance gains possible for future user-mode primitives. User-mode servers employing current "best practices" are used to establish a baseline for fastest possible user-mode performance. The tested user-mode servers employ several techniques to minimize data copies and reduce overhead of network event notification. The paper measures several kernel-mode Web servers, including servers based on a platform called "Adaptive Fast Path Architecture" (AFPA). The experiments are repeated for two different operating systems: Linux 2.3.51 and Windows 2000. In all cases, the CPU hardware, TCP/IP stack, network hardware and operating system are held constant.

The paper compares the performance of several user-mode and kernel-mode caching Web servers using different workloads. The results show a wide performance margin between the better performing user-mode and kernel-mode servers. The user-mode servers are shown effective in reducing memory copies and reads while also reducing scheduling overhead with efficient event notification mechanisms and single thread, asynchronous I/O implementations. The best of these efforts are still two to six times slower than the fastest achieved kernel-mode performance on the unmodified Linux and Windows 2000 operating systems tested using the same hardware. The results reveal significant potential to improve user-mode server performance.

The paper is organized as follows: Section 2 classifies Web server performance issues, describes current user-mode and kernel-mode approaches, and describes related work. Section 3 describes the Adaptive Fast Path Architecture, a platform for building kernel-mode network servers. Section 4 describes the methodology used to measure and analyze user-mode and kernel-mode Web server implementations. Section 5 reports and analyzes

the performance results for representative user-mode and kernel-mode Web servers. Finally, Section 6 draws conclusions from the performance analysis, and Section 7 makes recommendations regarding Web server design, and describes future work.

2 Web Server Performance Issues

Techniques to improve caching Web server performance address one or more of the following objectives: elimination of data copies and reads, reduction of scheduling and context switching overhead due to event notification, and reduction of overall communication overhead in the socket layer, TCP/IP stack, link layer, and network interface hardware.

2.1 Data Copies and Reads

Eliminating copies and reads for a Web transaction offers significant performance improvements for large responses. Data copies can be difficult to avoid in user-mode Web servers where the data to be sent resides in the file system cache. In cases where data is already mapped into the user-mode address space, BSD-style socket implementations will perform one or more copies before delivering the data to the network adapter. Even where data copies are eliminated, the additional overhead in reading the data to compute a checksum remains. The data copy problem is solved by providing a mechanism to send response data directly from the file system cache to the network interface. The checksum problem is solved either by precomputing and embedding the checksum in a Web cache object or by relying on network interface hardware to offload the checksum computation.

2.2 Event Notification

A second performance issue is minimizing the cost of event notification. We define event notification as the queuing of a client request by a server for response by a server task. In the case of a HTTP 1.0 request, the client request is formed by the arrival of a TCP SYN packet and subsequent data packets containing the request. The server must handle these two events with minimal overhead. First, it must complete the three-way TCP handshake started with the arrival of the first SYN packet. Second, it must receive the data forming the request, and read this data into a user-mode memory area. To handle multiple clients, the server supports concurrency either by assigning a single task from a pool of tasks to each

client request or by using asynchronous system calls to manage many requests with a few tasks.

Achieving efficient event notification requires a thread model with minimal scheduling overhead. The mapping between threads and requests is taxonomized by [5] as multiple process/thread (MP) or single process event driven (SPED). In the MP model, a server creates a new task for each new request. Because creating a new task can be time consuming, most MP servers reduce the overhead by pre-allocating a pool of tasks. However, pre-allocating a pool of tasks to avoid task creation still incurs unwanted scheduling overhead. Every request requires a reschedule to the task for that request. Apache [6] is the canonical example of the MP architecture. Ideally, the unnecessary scheduling inherent to the MP model is avoided in a design where a single task services requests on behalf of multiple clients.

In the SPED model, a few processes handle requests from multiple clients concurrently. The SPED model relies on asynchronous notification mechanism for notifying a server task of incoming network requests. For example, `select()` is the event notification mechanism commonly used on user-mode UNIX Web servers and I/O completion ports are commonly used on Windows 2000. Web servers such as Zeus [7], IIS [8] use a SPED model. Flash [5] also uses a SPED model for cached content, using only one thread for serving cache hits. The Windows 2000 APIs implementing zero copy data transfer and efficient event notification are described in [9].

2.3 Communication Code Path

A third performance issue is the overall communication code path through the socket layer, TCP/IP stack, link layer, and network interface. The socket layer is not necessarily tailored to the needs of Web servers. Researchers have modified existing socket APIs or implemented new APIs with Web servers in mind [10].

Some commercial operating systems provide interfaces specifically for Web servers. In particular, Windows NT provides interfaces eliminating redundant system calls: `AcceptEx()` and `TransmitFile()`. In addition to other benefits, these interfaces aggregate several system calls, reducing the code path between the Web server and TCP/IP stack. For example, `AcceptEx()` combines accepting a new connection, reading the request data, and obtaining peer address information, eliminating system calls and redundant socket layer code. Likewise, `TransmitFile()` combines reading data from the file system and writing header and data to the socket, also eliminating system calls.

In addition to socket layer optimizations, the TCP/IP stack has also been improved for Web server workloads. For example, TCP/IP implementations have been redesigned to efficiently manage short-lived connections [11]. Commercial operating systems have also been optimized for short-lived connections, improving management of TCP control blocks and sockets in the TIME_WAIT TCP/IP state.

The network interface hardware and corresponding driver are other key places for optimizations in the communication code path. For the purposes of this paper, we hold the network interface and driver implementations fixed when comparing servers on the same operating system. For completeness, it should be noted that network interfaces and their drivers can have a significant impact on performance. "Smart" network interface cards with on board processors are capable of coalescing interrupts, offloading fragmentation, checksum computation, and higher level TCP/IP processing such as connection establishment [12].

2.4 Current Approaches

User-mode Approaches

This paper analyzes several user-mode Web servers taking advantage of the performance enhancements described above. These user-mode Web servers rely heavily on the operating system to provide the primitives necessary to reduce data movement, limit event notification overhead, and minimize the communication code path. One approach is to optimize existing interfaces and their implementations. For example, implementations of `select()` and `poll()` have been improved by [13, 14] to reduce event notification overhead.

Another approach is to define completely new interfaces, building Web servers around these new interfaces. For example, new user-mode interfaces to eliminate memory copies and mitigate checksum computation include IO-Lite [15] and Windows NT's `TransmitFile()` API. IO-Lite provides a generic interface and mechanism to unify data management among operating system subsystems and user-mode servers. `TransmitFile()` provides the same performance effect in avoiding data copies, but is limited to sending files with prefix or suffix data from the file system cache. AIX implements a similar zero copy API called `sendfile()`. Linux provides a `sendfile()` API, but the implementation requires a data copy to move the data from the file system to the network stack. This paper analyzes the performance of IIS and Zeus, two production Web servers leveraging examples of these new APIs. Lastly, the paper describes

an additional SPED user-mode Web server for Windows 2000 and Linux called Howl. Howl is described in more detail in Section 5.3.

Kernel-mode Approaches

Kernel-mode Web servers have been implemented in the context of both production and experimental operating systems. Migration of services considered integral to a server's operation into the kernel is not a new idea. For example, most commercial operating systems include kernel-mode file servers. Delivery of static Web responses amounts to sending files on a network interface and does not require extensive request parsing. A kernel-mode Web server can fetch response data from a file system or kernel-managed Web cache. If the kernel-mode caching Web server determines that it cannot serve the request from its cache, it forwards it to a full-featured user-mode Web server.

Kernel-mode Web servers can be characterized according to the degree of their integration with the TCP/IP stack and whether responses are derived in a thread or interrupt context. Microsoft's Scalable Web Cache (SWC) [16] is tightly integrated with the Windows 2000 TCP/IP stack. By contrast, Linux's kHTTPd [17] uses socket interfaces in kernel-mode. Both SWC and kHTTPd handle response processing from kernel-mode threads. TUX [18] is another in-kernel Web server introduced by Red-Hat on Linux. Like kHTTPd, TUX uses a threaded model, but it offers greater features and performance. First, TUX caches objects in a pinned memory cache rather than using the file system. Second, TUX implements zero copy TCP send from this pinned memory cache and a checksum cache for network adapters without hardware support for offloading checksum computation.

Other Approaches

In addition to extending production operating systems, researchers have implemented specialized or new operating systems to experiment with Web server performance. The Lava hit-server [19] achieves cache performance limited only by memory bus bandwidth. While this paper holds the TCP/IP stack, network driver, and network hardware fixed, the hit-server focuses on network driver optimizations and a non-TCP transport protocol to minimize memory conflicts between the CPU and network controller for performance. The Cheetah Web server [20] is another example of a Web server designed with performance in mind on a new operating system. Rather than

extending production systems as described in this paper, the Cheetah Web server uses a specialized TCP stack on a research operating system called Exokernel [21]. The Exokernel approach demonstrates the performance possible when subsystems such as the network stack and file system are tightly integrated. The AFPA results in this paper appear consistent with the factor of three to six performance gain reported for Cheetah on the Exokernel operating system. However, the AFPA results use unmodified, production operating systems allowing direct comparisons with state of the art user-mode optimizations on Linux and Windows 2000.

3 AFPA

We now provide a brief overview of the Adaptive Fast Path Architecture (AFPA) [22], a software architecture for high-performance network servers. AFPA features:

- Support for a variety of application protocols.
- Direct integration with the TCP/IP protocol stack.
- A kernel-managed, zero copy cache.

3.1 Overview

AFPA is a flexible kernel-mode platform for high-performance network servers. The architecture is flexible in several ways. First, it can be applied to a variety of application protocols such as HTTP, FTP, LDAP, and DNS. Such protocols are implemented as AFPA modules. Second, it has been implemented on four platforms: Linux, Windows 2000, AIX, and OS/390. The latter three implementations have been incorporated into current IBM products, the first of which was released as the Netfinity Web Server Accelerator in 1998. The architecture was implemented on Linux and Windows 2000 solely as a kernel module. Third, it can be used as a caching server or an efficient layer 7 router. Fourth, it can be tightly integrated and co-located with a conventional user-level network server or implemented as a stand-alone front-end accelerator that offloads processing from a set of "backend" servers without requiring any modification to the conventional servers. Fifth, AFPA can be used to enforce quality of service [23]. This section focuses on AFPA application to Web servers.

Several factors contribute to AFPA's efficiency. These factors are now described in terms of data movement, event notification, and communication code path for the

Linux and Windows 2000 AFPA implementations. First, data copies and reads are eliminated, improving performance when sending larger responses. The data copies are avoided by passing references to pinned cache objects directly to the protocol stack. Reads are eliminated by avoiding checksum computation when sending cache objects as responses. On Windows 2000, checksum computation is off-loaded to the network interface hardware. On Linux, cache objects include pre-computed checksums.

Second, scheduling and context switching overhead in responding to TCP/IP events is significantly reduced or eliminated using AFPA. AFPA parses requests on the same software interrupt on which TCP/IP processing occurs. AFPA then sends corresponding responses from the same interrupt context or queues the response for sending in a thread context. In implementations where responses are derived from software interrupt context, no scheduling or context switching overhead is incurred. As shown in Section 5, responding from software interrupt provides better performance, but responses must reside in pinned memory. The AFPA module can also use a thread-based configuration where responses are sent from a thread context. This approach mitigates livelock problems inherent to the software interrupt approach [24]. A hybrid approach has also been implemented. Requests for content not currently pinned are processed on software interrupt, but unpinned responses are sent from a kernel-mode thread context where page faults are tolerated.

Third, the overall communication overhead incurred in AFPA implementations is less than a user-mode server relying on a socket API. AFPA interfaces directly with TCP/IP by overloading TCP/IP events with HTTP specific processing. These events include connection establishment (SYN), data arrival, and disconnection (FIN). In overloading these events, AFPA drives a state machine associated with the protocol modules such as HTTP. Direct integration with TCP avoids the queueing and descriptor management incurred using a socket API.

3.2 State Machine

Using a state machine interface, it is possible to implement a variety of protocols in the AFPA framework. Protocol specific code for a given application such as Web serving is encapsulated in an AFPA module.

Each AFPA module is partitioned into three components: state, program actions, and control elements. AFPA manages three types of internal state data for each module instance: global data, connection data, and request data. Global data is data independent of each connec-

tion and request. Connection data is information that changes on a per connection basis. Request data is information that changes per request within the same connection. State functions are either written by the module developer, exist in another AFPA module's library of state functions, or are intrinsic to AFPA as a general service. State functions indirectly manage state transitions through a separately managed table. The layer of indirection permits state functions to be reusable and reconfigurable.

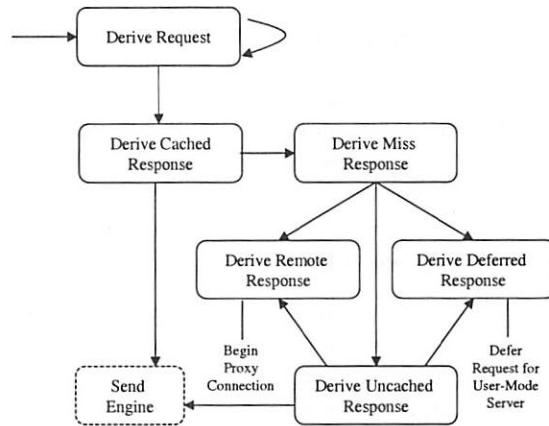


Figure 1: AFPA HTTP State Machine

A simplified version of the state machine used by the AFPA HTTP module is shown in Figure 1. The state machine illustrates how the HTTP module can be used as a standalone Web cache, work in conjunction with a user-mode Web server, or act as a front end Web cache for one or more back end Web servers. The module implements a function corresponding to each of the following client connection states:

- **Derive Request**

This function is invoked when new data is received on a connection. When sufficient data has arrived to complete a well-formed request, the function computes the parsed request, storing it in the connection data structure.

- **Derive Cached Response**

This function uses the parsed request in the connection data structure to perform a key-based lookup in the AFPA cache. If successful, the cache object is sent. If unsuccessful, the connection transitions to 'Derive Miss Response' state.

- **Derive Miss Response**

This function is invoked when a response is not found in the AFPA cache. The HTTP module either performs necessary file I/O to create a new cache object (Derive Uncached Response), passes the request to a local user-mode Web server (Derive Deferred Response), or sends the request to a remote Web server (Derive Remote Response).

- **Derive Uncached Response**

This function creates a cache object and response header. The file is either read into the newly created cache object or a reference to the file system cache is created depending on the AFPA implementation. Once created, the cache object is sent to the client. Any errors creating the cache object may result in sending request to another Web server or generating an error response.

- **Derive Remote Response**

This function routes the parsed request to a remote server. In versions of AFPA modules acting as content-based routers, this function expands to an alternate state machine for managing connections to other Web servers.

- **Derive Deferred Response**

This function routes the parsed request to a user-mode server which takes control of the connection.

- **Send Engine**

The Send Engine is a function invoked by the Derive Cached Response or Derive Uncached Response states. The send engine sends the cache object. Persistence, serialization, and fragmentation of large requests are managed by AFPA. This function is exported by the AFPA runtime system.

3.3 TCP/IP Integration

A key aspect of AFPA is its close integration with the TCP/IP protocol stack. Layer 7 protocols are processed on the same software interrupt on which TCP/IP input processing occurs. To achieve this, AFPA relies on the ability to extend the TCP/IP stack through callback mechanisms. AFPA cache objects use native TCP/IP data structures such as BSD mbufs, Linux skbuffs, or Windows 2000 MDLs (lists of chained page frames used to describe buffers) [25].

3.3.1 Linux TCP/IP Integration

AFPA on Linux intercepts events in the TCP/IP stack without kernel modification. The Linux kernel socket structure contains function pointers which are invoked whenever the state of the connection changes, inbound data is queued on the socket, or outbound data is removed from the socket queue. AFPA on Linux replaces the data arrival hook with its own. When the first data packet arrives on the socket, the main AFPA hook gets called from within the network bottom half (i.e. software interrupt handler). AFPA then parses the request packet, looks up the cache object, queues the response in the socket output queue, and sends the response. If the request does not fit entirely into the first packet, AFPA creates a connection context which it retrieves when the next packet arrives and parses the request.

To manage sending data, AFPA on Linux rewrites the skbuff free hook, which is called whenever a network buffer is freed. This hook is used to send responses in 64 kB chunks. The last packet of a chunk is flagged. When the AFPA hook detects that last packet of a chunk, the AFPA hook sends the next chunk. When a request is not found in the cache, the request is queued and picked up by a service kernel thread. Tight synchronization has to be provided between the thread and the software interrupt which are competing for the socket's accept queue.

AFPA on Linux allocates a number of 128 KB blocks of pinned memory which are managed by AFPA's own memory allocator. When APFA creates a cache object, it opens the corresponding file and reads it into Ethernetframe-sized buffers. AFPA allocates space in each buffer for TCP, IP, and Ethernet headers. It reserves additional space in the first buffer for HTTP headers. When APFA receives a request, it fills in the HTTP header and queues each frame associated with the object for transmission. For large responses, AFPA queues frames in 64 KB chunks. If an additional request is received for a cache object that is in the process of being sent, AFPA makes an additional copy of the buffer (i.e. Ethernet frame).

3.3.2 Windows 2000 TCP/IP Integration

On Windows, AFPA interacts with the TCP stack using the Transport Driver Interface (TDI) [25]. TDI is an interface, defined by Microsoft, by which kernel-mode "clients" interact with protocol drivers such as TCP/IP. TDI defines a set of client requests, including accept, connect, disconnect, send, and receive. It also defines a set of callback routines, each associated with a network event such as connection establishment, discon-

nection, and reception. TDI allows but does not require a client to register a callback routine for any event per connection end point. TDI uses an asynchronous model. Each request has an associated client-specified completion routine that is invoked when the request completes (whether synchronously or asynchronously). Client-registered callback routines are invoked asynchronously as well.

When a TCP SYN packet arrives on a port to which AFPA is bound, TDI invokes AFPA's connect event handler. This routine allocates an AFPA connection structure, in which application-specific information associated with the connection is stored, then builds and returns an accept request. The completion routine for the accept request simply cleans up in case of error. Arrival of request data from the client causes AFPA's receive event handler to be invoked.

On Windows 2000, AFPA reads cache objects from the file system, pins them in memory, and passes them to the TCP stack in 64 KB chunks. Each chunk is represented by an entry in the cache object's pin array. When AFPA creates a cache object, it opens the corresponding file. AFPA initializes each entry in the pin array to indicate that the corresponding chunk has not yet been read or pinned. When it sends a cache object, AFPA sequentially traverses the object's pin array. If a chunk is currently pinned, its pin count is incremented, and it is sent immediately in the context of a software interrupt. If the chunk is not pinned, the request must be queued to a pool of worker threads because the Windows 2000 memory architecture does not permit access to unpinned memory from software interrupts. The send completion routine decrements the pin count (which acts as a reference count) for the current chunk and repeats the process for the next chunk.

3.4 Cache Architecture

A complete description of the AFPA cache is beyond the scope of this paper. We do, however describe several aspects particularly relevant to response processing. First, AFPA cache objects contain mutable header data, immutable header data, and data payload. On Windows 2000, the cache object is represented as an MDL. Cache objects do not include precomputed checksums because Windows 2000 supports a number of network adapters that offload checksum computation as well as payload fragmentation. On Linux, the cache object is presented as a list of skbuff structures with precomputed checksums. This architecture allows zero copy send operations, not supported by the Linux 2.2 kernel. Second, AFPA cache objects are opaque data types which can be

backed by any number of memory systems. For example, it is possible to implement AFPA cache objects using Windows 2000 support for x86 PAE (Physical Address Extensions) mode and manage a cache of up to 64 GB in size. It is also possible to back AFPA cache objects directly with the file system cache, thereby leveraging the system cache for selected cached files. Third, the cache architecture supports a hybrid approach to handling responses at software interrupt or kernel thread. This allows a two level cache where frequently accessed small files remained pinned, allowing them to be delivered in a software interrupt context, while larger and less frequently accessed files are served using threads. Finally, cache objects are divided into fragments. Each fragment can be pinned and passed to the TCP/IP stack independently.

4 Experimental Methodology

In this section, we present experimental methodology used to compare user-mode Web server performance with kernel-mode implementations based on the AFPA framework described in the previous section. The goal of the methodology is to establish a baseline for user-mode performance and compare the best performing user-mode approaches with Web servers based on AFPA. We compare several user-mode and kernel-mode HTTP servers: Apache 1.3.9 [6], Zeus 3.3.5 [7], IIS 5.0 [8], and two experimental Web servers referred to as Howl. We also consider other kernel-mode Web servers: kHTTPd [17] and TUX [18] on Linux, and Microsoft's SWC 2.0 [16].

4.1 Workload

We use two different synthetic workloads for our experiments: SPECWeb96 [26] and WebStone 2.5 [27]. SPECWeb96 was the first standard HTTP benchmark. The SPECWeb96 working set comprises files that range in size from 100 bytes to 900 kB, where small files are referenced more often than large files (50% of the total number of requests reference files smaller than 10 kB). In addition, the SPECWeb96 working set scales with the expected server throughput. In all of our experiments, the entire working set fit into the server's RAM, thus avoiding any performance distortion due to disk accesses.

SPECWeb96 has been superseded by SPECWeb99 as the industry-accepted Web serving benchmark. SPECWeb99 exercises HTTP/1.1 features, such as persistent connections, and includes requests for dynami-

cally generated pages.

Although SPECWeb96 does not take into account some aspects of current HTTP workloads (e.g. no persistent connections, no dynamic content), it is well suited for measuring static file serving performance, which is the main purpose of our performance evaluation. Furthermore, large HTTP sites often use several servers that are partitioned into groups serving different types of content such as static files, user logins, and databases. The static content servers are likely to experience workloads similar to the SPECWeb96 workload. Finally, the SPECWeb96 execution guidelines are sufficiently strict as to allow meaningful comparison of independently reported results.

The results presented here do not meet SPECWeb96 reporting guidelines and are not certified SPECWeb96 results. The SPECWeb96 benchmark was executed for the largest workload corresponding to the reported result rather than ten evenly spaced lower throughput workloads as required by SPECWeb96 for reporting purposes. This does not affect the results reported in this paper.

WebStone is another HTTP server benchmark. Unlike SPECWeb96, it allows a user to change the workload characteristics, making it easier to identify performance bottlenecks for given file sizes. For WebStone, our workload consists of fixed-size files, ranging from 64 bytes to 1 MB. The file size is varied in each test.

4.2 Test Environment

Experiments were performed on two operating systems: Windows 2000 Advanced Server (build 2195) and Red-Hat Linux 6.1 with a Linux 2.3.51 kernel. One server, TUX, which does not run on a Linux 2.3.51 kernel, was run on a Linux 2.4.0 kernel instead. AFPA on Windows 2000, IIS, and SWC were run on Windows 2000. AFPA on Linux, kHTTPD, TUX, Zeus, and Apache were run on Linux. To quantify the benefit of serving responses in a software interrupt context, a version of AFPA that does not include this optimization and instead serves all responses using kernel threads was implemented.

All experiments were performed on the same server hardware: an IBM Netfinity 7000 M10 with four 450 Mhz Pentium II Xeon processors, 4 GB of RAM and four Alteon ACEnic gigabit Ethernet adapters. The server hardware has two 33 Mhz PCI buses (one 32 bit and one 64 bit). Each PCI bus had two gigabit Ethernet adapters. Distributing these adapters over the two PCI buses was necessary to maximize the bandwidth of the memory bus. For all experiments, only one of the server's four CPUs were used. The presence of three empty CPU socket

ets does not interfere with the uniprocessor experiments. Ten client machines were used to generate load. The clients were IBM Intellistation Z-Pro systems with two 450 Mhz Pentium II Xeon processors, 256 MB RAM), and a single Alteon ACEnic gigabit Ethernet adapter. The clients ran RedHat Linux 6.1 and were connected to the server via a pair of Alteon ACEswitch 180 gigabit Ethernet switches.

The Netfinity 7000 M10 supports up to 280 MB/s memory to memory bandwidth based on timing `memcpy()`. In practice, the tested Netfinity hardware is at most capable of 200 MB/s bandwidth from main memory to the PCI buses. Including TCP/IP headers, HTTP request, and HTTP response, the maximum possible SPECWeb96 result is 11,400 requests per second.

All experiments were run using 9000 byte (jumbo) Ethernet frames. We chose jumbo Ethernet frames rather than standard 1500 byte Ethernet frames since it allowed our SPECWeb96 results to be compared with officially published results [26]. Limited experiments using standard Ethernet frames did not reveal in any significant difference in the performance trends seen with 9000 byte frames.

We note the following limitations of our test methodology. All experiments were performed with the same limited number of client machines. Our results focus almost entirely on uniprocessor rather than multiprocessor servers. Experiments were performed solely with non-persistent connections. Our analysis is constrained to static content only. Finally, results are reported only for the Linux and Windows 2000 operating systems running on the same Intel processor.

4.3 Performance Tuning

On the server side, Linux and Windows 2000, as well as each individual Web server, were tuned to achieve maximum performance. To this end, we used tuning parameters provided with submitted SPECWeb96 results. For servers that support time-to-live values for cached objects, we set the timeout to eliminate cache invalidations. This ensures we achieve a 100% hit rate.

5 Performance Analysis

Two benchmarks are used to compare the user-mode and kernel-mode Web servers. SPECWeb96 is used for comparing workloads with mixed file sizes. Webstone is used to compare performance for fixed file sizes.

5.1 SPECWeb96 workload

The results for the SPECWeb96 workload are presented in Figure 2. Results are presented for the following user-mode Web servers: Apache (Linux), Zeus (Linux), IIS (Windows 2000). Results are also presented for the following kernel-mode Web servers: kHTTPd (Linux), TUX (Linux), SWC (Windows 2000), AFPA on Linux, and AFPA on Windows 2000. Table 1 enumerates and describes the Web servers tested on Linux and Windows 2000.

	architecture	cache	0 copy	direct TCP
Apache	MP/user	filesystem	no	no
Zeus	SPED/user	filesystem	no	no
IIS	SPED/user	filesystem	yes	no
kHTTPd	SPED/kernel	filesystem	no	no
TUX	SPED/kernel	memory	yes	no
SWC	SPED/kernel	fs or mem	yes	yes
AFPA	softint/kernel	fs or mem	yes	yes

Table 1: Web Server Characteristics

The “architecture” column describes servers as MP, SPED, or softint (software interrupt) as defined in Section 2.2 and kernel-mode or user-mode. The “cache” attribute defines whether the Web server’s cache is backed by the file system, memory, or both. The “0 copy” column indicates whether or not the Web server performs a copy to send a cache object. The “direct TCP” column indicates whether or not the Web server is directly integrated with the TCP/IP stack or uses the socket layer.

Kernel-mode servers appear to have a factor of three performance advantage over production user-mode servers. On Linux, the fastest user-mode server measured (Zeus) is 3.6 times slower than the fastest kernel-mode server (AFPA). On Windows 2000, the fastest user-mode server measured (IIS) is 3 times slower than the best kernel-mode server (AFPA). Overall, the fastest kernel-mode implementation (AFPA on Linux) is 3.5 times faster than the best performing user-mode implementation measured (IIS on Windows 2000).

The slowest user-mode server in the SPECWeb96 results was Apache. This is consistent with other published results [5, 28]. Apache seems to be penalized by a significant process scheduling overhead. Note, however, that Apache 1.3.9 does not feature a memory-based static content cache; it uses the file system cache. Among other optimizations, adding a memory based cache to Apache reportedly increases its performance by 70% on Linux [29] which would bring Apache in line with IIS and Zeus.

As mentioned before, both IIS and Zeus employ SPED architectures. Although Linux does not feature zero copy send, Zeus was on par with IIS. This somewhat contradicts previous attempts at comparing user-mode Linux and Windows Web servers [28]. These earlier results were, however, obtained with Apache which exhibits lower performance than Zeus.

In comparing kernel-mode servers, we found kHTTPd to be relatively slow on the SPECWeb96 workload compared to other kernel Web servers. In fact, kHTTPd achieves nearly the same SPECWeb96 result (2893) as Zeus (2843). kHTTPd has limited performance for two reasons. First, kHTTPd requires one copy to send the response. This is unavoidable due to kHTTPd’s reliance on the file system as a cache. The Linux file system interfaces lack a zero copy mechanism to send file system data. Second, kHTTPd uses the kernel-mode version of the Linux socket interface rather than interfacing directly with the TCP/IP stack. Therefore, kHTTPd’s performance is not significantly different than Linux user-mode Web servers, which are also forced by Linux to use a one-copy send and a socket interface.

TUX offers nearly twice the performance of kHTTPd, due primarily to its pinned memory cache and zero copy send implementation. TUX and kHTTPd have otherwise similar architectures for serving static content out of main memory. They both use the socket API from kernel threads for sending objects.

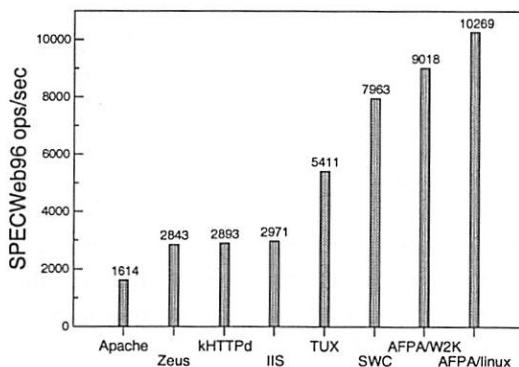


Figure 2: SPECWeb96 results

The results show that AFPA on Linux achieves the fastest performance of the tested servers. The SPECWeb96 result of 10,269 represents over 1.2 Gb per second server throughput. This amounts to 90% of the hardware capacity as described in section 4.2. Therefore, AFPA on Linux is a reasonable performance target for evaluating user-mode optimizations.

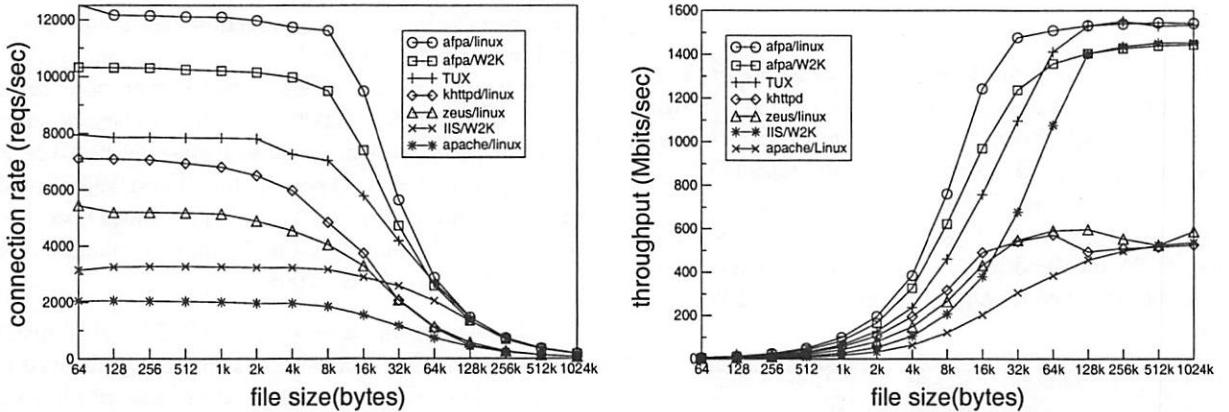


Figure 3: Fixed file size performance results

5.2 Analyzing a Fixed File Size Workload

We next studied how performance varied as a function of response size using a set of file sizes ranging from 64 bytes to 1 MB. The experiments were performed using the Webstone benchmarking tool. The connection rates and delivered bandwidths are reported in Figure 3. For small files, request latency was the dominant performance factor.

Linux vs. Windows 2000

For 64 bytes files, AFPA on Linux was 21% faster (12,522 requests per second) than AFPA on Windows 2000 (10,321 requests per second). In addition to obtaining the requests per second, we also used the Intel processor performance counters to measure several metrics under the same workload. We found two metrics significantly different between AFPA on Linux and Windows 2000: the number of instructions executed and instruction TLB misses. Because the average cycles per instruction were nearly identical for both cases, we conclude that the instruction count is a useful metric for comparing the two implementations. In addition, both AFPA implementations used the exact same source code to implement the HTTP and caching logic. This implies that any differences between the two would have to be limited to the interfaces used to integrate AFPA in the TCP/IP stack, the TCP/IP stack itself, and network driver. AFPA on Linux executed 19% fewer instructions than AFPA on Windows 2000 (26,000 versus 31,000 instructions per request). We also find that the number of instruction TLB misses was ten per request on Windows 2000 versus zero per request on Linux. The Linux kernel, TCP/IP stack,

and kernel modules are stored entirely in non-pageable 4 MB pages, so it does not experience any instruction TLB misses. Only the Windows 2000 kernel is mapped using 4 MB pages; the TCP/IP stack is not.

Thread vs. Software Interrupt

Using the Pentium performance counters, we also compared the software interrupt-version of AFPA with the threaded version of AFPA on Windows 2000. For 64 byte files the software interrupt version was 12% faster than the threaded version (10,321 versus 9,209 requests per second). This closely matches the difference in the number of instructions executed. This difference corresponds to the overhead of queueing/dequeueing work items and scheduling the thread.

Effect of File Size

For large files, performance is determined primarily by the speed at which the server can move data to the network. As file size increases, the operating system overhead for user-mode servers accounts for less and less in the overall cost of processing requests. This is because processing latency is completely amortized for large files. For example, Apache lags behind the other Web servers for performance on small files, but is just as good as other user-mode Linux servers for large files. On Windows 2000, IIS performs as well as AFPA for files 128 kB and larger, while it is 3.27 times slower than AFPA for 64 byte files.

For user-mode Web servers, IIS was slower than Zeus for files smaller than 32 kB, but for larger files gained an

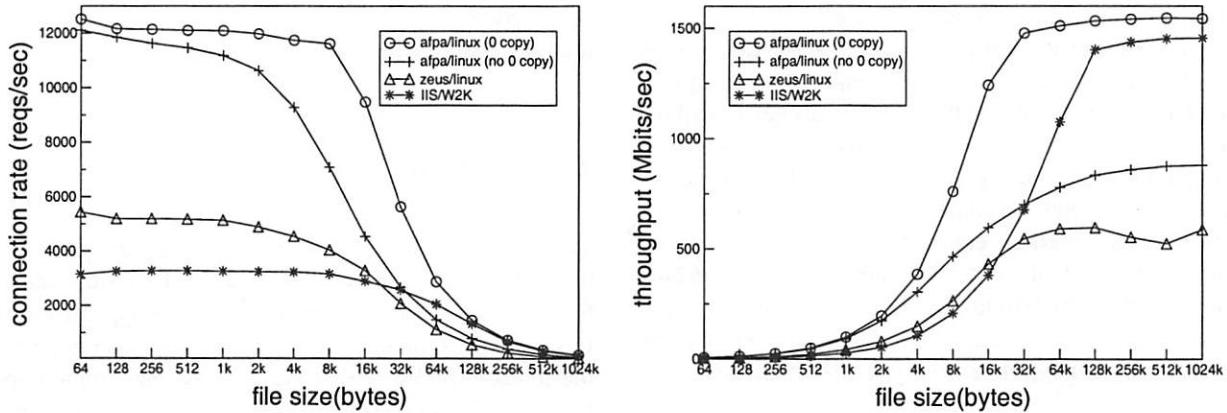


Figure 4: Efficiency of zero copy TCP send

advantage from having a zero copy send interface. The importance of a zero copy TCP send is further emphasized on the throughput graph. There is almost a three-fold performance difference between Web servers using zero copy send interfaces and those using a one-copy send interface. TUX achieved three times the throughput of kHTTPd for large files. We also ran a modified version of AFPA on Linux that does not use the AFPA zero copy architecture. Its throughput on large files was half that of the zero copy version.

Jumbo Frames

Another interesting result is the nearly flat connection rate for transfers less than 8 kB. Even with jumbo frames enabled one would expect a more significant decrease in the connection rate. It appears that the Alteon firmware is optimized for bulk data transfers rather than fast connection set-up. We ran some tests using a single client thread requesting 1 kB transfers on Alteon, Intel gigabit, and Intel 100 Mb adapters. This configuration measures connection latency. We found the Alteon adapter to be between two and three times slower than the Intel gigabit and 100 Mb adapters, respectively. The high connection set-up cost on the Alteon adapter most probably accounts for the flat connection rate.

Zero copy TCP send

In order to evaluate the performance gain of a zero copy send interface in the TCP/IP stack, we ran a modified version of AFPA on Linux that does not use the AFPA zero copy cache architecture. In this version, network buffers are allocated through the standard Linux

`sock_wmalloc()` primitive; file data is copied from the AFPA cache into network buffers and checksummed before being sent. Figure 4 summarizes the performance of these two implementations plus Zeus on Linux (which does not use zero copy sends) and IIS on Windows 2000 (which does zero copy sends through the `TransmitFile()` API).

As expected, the performance advantage of a zero copy send interface increased with the file size. It is important to note, however, that the benefits of a zero copy interface can be seen for relatively small files. For 4 kB files the performance difference is 25% and then grows to 111% for 32 kB files.

For full efficiency, a zero copy send interface also requires a network adapter with outbound packet checksumming capability (such as the Alteon adapter used in our test bed) in order to avoid reading the data to checksum it.

5.3 Howl

In defining the best possible user-mode performance it's important to not rely solely on commercial user-mode examples for performance analysis. To that end a user-mode Web server was implemented using the best practices for performance on Linux and Windows 2000. This user-mode Web server is referred to as Howl. Howl is an attempt at estimating the maximum performance that can be achieved by a user-mode server on current versions of Linux and Windows 2000 using standard APIs. On Linux, Howl is a simple loop executing four system calls to process an HTTP request: `accept()`, `read()`, `write()`, and `close()`. It uses a user-mode version of the AFPA cache for storing responses (with pre-

generated HTTP response headers). Howl offers very limited functionality and performance (no logging, only one request is processed at a time). Howl is a test case. It is not intended for general use. But under the assumption that (i) all requests fit into the first data packet, (ii) all requests hit in the cache, (iii) all responses can be sent with a non blocking write (by configuring a sufficiently large socket buffer), and (iv) consecutive client requests do not block the server task receiving the request, Howl has performance close to the best achievable using the standard Linux APIs. On Windows 2000, Howl uses `AcceptEx()`, `TransmitFile()`, and I/O Completion ports to achieve best possible user-mode performance.

Compared to the user-mode results shown in Figure 3, Howl on Linux is 32% faster than Zeus and Howl on Windows is 21% faster than IIS for 64 byte files. Likewise, for 1 kB files, Howl on Linux is 29% faster than Zeus and Howl on Windows is 16% faster than IIS. Given that Zeus and IIS are production-level web servers, it is not surprising to marginally improve upon their performance with minimal prototypes such as Howl. However, the 16% to 29% performance improvement using such prototypes for 1 kB files is significantly smaller the 235% to 312% gap between these servers and their kernel-mode counter parts. Given the size of this gap, further significant performance improvements appear unlikely without new user-mode APIs and operating system modifications.

6 Conclusion

The paper showed the performance of several highly optimized user-mode Web servers, comparing these servers to multiple kernel-mode Web servers while holding TCP/IP and network driver implementations fixed. The paper concludes that the best performing user-mode Web servers are at least two times slower than the faster kernel-mode server on the same hardware and unmodified operating system. The best kernel-mode results were achieved using a software interrupt approach where responses are sent on software interrupt (Linux bottom half handler or Windows 2000 deferred procedure call) rather than a separately scheduled thread. The results showed that software interrupt based kernel-mode servers perform 10% to 20% better than Overall, the best performing Web servers share three attributes. First, they use a zero copy interface between cache and network without TCP checksum computation to efficiently serve responses greater than 4 kB in size. Second, these servers use an efficient event notification mechanism to serve responses less than 4 kB in size with minimal scheduling overhead. Third, these servers minimize communication code path using new socket APIs or eliminating the socket layer altogether.

7 Future Work

The results motivate future work to close the gap with the kernel-mode approaches described in this paper. First, the software interrupt kernel-mode approach suffers from the problems described by [24]. Second, even the thread-based kernel-mode approach has limited applicability beyond simple caching. While it's possible to embed application-specific code in the kernel, the approach is awkward and leads to a paradigm where the benefits of address spaces are lost.

Future work will focus on support for user-mode servers to achieve kernel-mode performance without implementing application-specific code in the kernel. In particular, the performance advantages of a kernel-mode approach might be amortized by batching multiple layer 7 requests before indicating them to a user-mode server. Likewise, response processing might also be aggregated over multiple responses before indicating those responses back to the kernel. Future work will explore batching techniques to amortize user-mode overhead using a kernel-mode request/response engine and cache.

References

- [1] Akamai Technologies, Inc. Akamai freeflow service. <http://www.akamai.com/service/network.html>.
- [2] TimesTen Performance Software. Timesten front-tier. <http://www.timesten.com>.
- [3] Jim Challenger, Arun Iyengar, and Paul Danzig. A scalable system for consistently caching dynamic Web data. In *Proceedings of IEEE INFOCOM'99*, March 1999.
- [4] Alec Wolman, Geoff Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry Levy. On the scale and performance of cooperative Web proxy caching. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 16–31, December 1999.
- [5] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, June 1999.
- [6] The Apache Group. Apache http server project. <http://www.apache.org>.
- [7] Zeus Technology Ltd. Zeus web server. <http://www.zeus.com>.

- [8] Microsoft Corporation. Internet information services features. <http://www.microsoft.com/windows2000/guide/server/features/web.asp>.
- [9] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. High performance Web servers on Windows NT: Design and performance. In USENIX, editor, *The USENIX Windows NT Workshop 1997, August 11–13, 1997. Seattle, Washington*, pages 149–149, Berkeley, CA, USA, August 1997. USENIX.
- [10] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Better operating system features for faster network servers. In *Proceedings of the Workshop on Internet Server Performance (held in conjunction with ACM SIGMETRICS '98)*, Madison, WI, June 1998.
- [11] Jeffrey C. Mogul. Operating systems support for busy internet servers. Technical Report Technical Note TN-49, Digital Western Research Laboratory, Palo Alto, CA., May 1995.
- [12] Alacritech. <http://www.alacritech.com>.
- [13] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Usenix Annual Technical Conference*, pages 253–265, 1999.
- [14] Niels Provos and Chuck Lever. Scalable network I/O in Linux. In *Proceedings of the FREENIX Track: 2000 USENIX Annual Technical Conference (FREENIX-00)*, pages 109–120, Berkeley, CA, June 18–23 2000. USENIX Association.
- [15] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. IO-lite: A unified I/O buffering and caching system. In *Operating Systems Design and Implementation (OSDI '99)*, pages 15–28, 1999.
- [16] Microsoft Corporation. Installation and performance tuning of microsoft scalable web cache (swc 2.0). <http://www.microsoft.com/technet/iis/swc2.asp>.
- [17] Arjan van de Ven. kHTTPD Linux http accelerator. <http://www.fenrus.demon.nl>.
- [18] Ingo Molnar. Answers from planet TUX: Ingo Molnar responds. <http://slash-dot.org/articles/00/07/20/1440204.shtml>.
- [19] Jochen Liedtke, Vsevolod Panteleenko, Trent Jaeger, and Nayeem Islam. High-performance caching with the lava hit-server. In *Proceedings of the USENIX 1998 Annual Technical Conference*, pages 131–142, Berkeley, USA, June 15–19 1998. USENIX Association.
- [20] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Héctor M. Briceño, Russel Hunt, and Thomas Pinckney. Fast and flexible application-level networking on exokernel systems. Technical Report CMU-CS-00-117, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA., mar 2000.
- [21] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 52–65, New York, October 5–8 1997. ACM Press.
- [22] Elbert C Hu, Philippe A Joubert, Robert B King, Jason LaVoie, and John M Tracey. Adaptive Fast Path Architecture. *to appear in IBM Journal of Research and Development*, April 2001.
- [23] Thiemo Voigt, Renu Tewari, Douglas Freimuth, and Ashish Mehra. Differentiation in overloaded web servers. In *Proceedings of the USENIX 2001 Annual Technical Conference*, June 2001.
- [24] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Transactions on Computer Systems*, 15(3):217–252, August 1997.
- [25] David Solomon and Mark Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 2001.
- [26] The Standard Performance Evaluation Corporation. Specweb96 benchmark. <http://www.spec.org/osg/web96>.
- [27] Mindcraft Inc. Webstone - the benchmark for web servers. <http://www.mindcraft.com/webstone>.
- [28] Mindcraft Inc. Open benchmark: Windows NT Server 4.0 and Linux. <http://www.mindcraft.com/whitepapers/openbench1.html>.
- [29] SGI. Accelerating apache. <http://www.oss.sgi.com/apache>.

Kernel Mechanisms for Service Differentiation in Overloaded Web Servers

Thiemo Voigt*

Swedish Institute of Computer Science

thiemo@sics.se

Renu Tewari

IBM T.J. Watson Research Center

tewarir@us.ibm.com

Douglas Freimuth

IBM T.J. Watson Research Center

dmfreim@us.ibm.com

Ashish Mehra

iScale Networks

ashish@iscale.net

Abstract

The increasing number of Internet users and innovative new services such as e-commerce are placing new demands on Web servers. It is becoming essential for Web servers to provide performance isolation, have fast recovery times, and provide continuous service during overload at least to preferred customers. In this paper, we present the design and implementation of three kernel-based mechanisms that protect Web servers against overload by providing admission control and service differentiation based on connection and application level information. Our basic admission control mechanism, *TCP SYN policing*, limits the acceptance rate of new requests based on the connection attributes. The second mechanism, *prioritized listen queue*, supports different service classes by reordering the listen queue based on the priorities of the incoming connections. Third, we present *HTTP header-based connection control* that uses application-level information such as URLs and cookies to set priorities and rate control policies.

We have implemented these mechanisms in AIX 5.0. Through numerous experiments we demonstrate their effectiveness in achieving the desired degree of service differentiation during overload. We also show that the kernel mechanisms are more efficient and scalable than application level controls implemented in the Web server.

1 Introduction

Application service providers and Web hosting services that co-host multiple customer sites on the same server cluster or large SMP are becoming increasingly common in the current Internet infrastructure. The increasing growth of e-commerce on the web means that any server down time that affects the clients being serviced will result in a corresponding loss of revenue. Additionally, the unpredictability of flash crowds can overwhelm a hosting server and bring down multiple customer sites simultaneously, affecting the performance of a large number of clients. It becomes essential, therefore, for hosting services to provide performance isolation and continuous operation under overload conditions.

Each of the co-hosted customer sites or applications may have different quality-of-service (QoS) goals based on the price of the service and the application requirements. Furthermore, each customer site may require different services during overload based on the client's identity (preferred gold client) and the application or content they access (e.g., a client with a buy order vs. a browsing request). A simple threshold based request discard policy (e.g., a TCP SYN drop mode in commercial switches/routers discards the incoming, oldest or any random connection [1]) to delay or control overload is not adequate as it does not distinguish between the individual

*This work was partially funded by the national Swedish Real-time Systems Research Initiative (ARTES). This work was done when the author was visiting the IBM T.J. Watson Research Center.

QoS requirements. For example, it would be desirable that requests of non-preferred customer sites be discarded first. Such QoS specifications are typically negotiated in a service level agreement (SLA) between the hosting service provider and its customers. Based on this governing SLA, the hosting service providers need to support service differentiation based on client attributes (IP address, session id, port etc.), server attributes (IP address, type), and application information (URL accessed, CGI request, cookies etc.).

In this paper, we present the design and implementation of kernel mechanisms in the network subsystem that provide admission control and service differentiation during overload based on the customer site, the client, and the application layer information.

One of the underlying principles of our design was that it should enable “early discard”, i.e., if a connection is to be discarded it should be done as early as possible, before it has consumed a lot of system resources [2]. Since a web server’s workload is generated by incoming network connections we place our control mechanisms in the network subsystem of the server OS at different stages of the protocol stack processing. To balance the need for early discard with that of an informed discard, where the decision is made with full knowledge of the content being accessed, we provide mechanisms that enable content-based admission control.

Our second principle was to introduce minimal changes to the core networking subsystem in commercial operating systems that typically implement a BSD-style stack. There have been prior research efforts that modify the architecture of the networking stack to enable stable overload behavior [3]. Other researchers have developed new operating system architectures to protect against overload and denial of service attacks [4]. Some “virtual server” implementations try to sandbox all resources (CPU, memory, network bandwidth) according to administrative policies and enable complete performance isolation [5]. Our aim in this design, however, was not to build a new networking architecture but to introduce simple controls in the existing architecture that could be just as effective.

The third principle was to implement mechanisms that can be deployed both on the server as well as outside the server in layer 4 or 7 switches that perform load balancing and content based routing

for a server farm or large cluster [6]. Such switches have some form of overload protection mechanisms that typically consists of dropping a new connection packet (or some random new connection packet) when a load threshold is exceeded. For content-based routing the layer 7 switch functionality consists of terminating the incoming TCP connection to determine the destination server based on the content being accessed, creating a new connection to the server in the cluster, and splicing the two connections together [7]. Such a switch has access to the application headers along with the IP and TCP headers. The mechanisms we built in the network subsystem can easily be moved to the front-end switch to provide service differentiation based on the client attributes or the content being accessed.

There have been proposals to modify the process scheduling policies in the OS to enable preferred web requests to execute as higher priority processes [8]. These mechanisms, however, can only change the relative performance of higher priority requests; they do not limit the requests accepted. Since the hardware device interrupt on a packet receive and the software interrupt for packet protocol processing can preempt any of the other user processes [3] such scheduling policies cannot prevent or delay overload. Secondly, the incoming requests already have numerous system resources consumed before any scheduling policy comes into effect. Such priority scheduling schemes can co-exist with our controls in the network subsystem.

An alternate approach is to enable the applications to provide their individual admission control mechanisms. Although this achieves application level control it requires modifications to existing legacy applications or specialized wrappers. Application controls are useful in differentiating between different clients of an application but are less useful in preventing or delaying overload across customer sites. More importantly, various server resources have already been allocated to a request before the application control comes into effect, violating the early discard policy. However, the kernel mechanisms can easily work in conjunction with application specific controls.

Since most web servers receive requests over HTTP/TCP connections, our controls are located in three different stages in the lifetime of a TCP connection.

- The first control mechanism, *TCP SYN policing*, is located at the start of protocol stack processing of the first SYN packet of a new connection and limits acceptance of a new TCP SYN packet based on compliance with a token bucket based policer.
- The next control, *prioritized listen queue*, is located at the end of a TCP 3-way handshake, i.e., when the connection is accepted and supports different priority levels among accepted connections.
- Third, *HTTP header-based connection control*, is located after the HTTP header is received (which could be after multiple data packets) and enables admission control and priority values to be based on application-layer information contained in the header e.g., URLs, cookies etc.

We have implemented these controls in the AIX 5.0 kernel as a loadable module using the framework of an existing QoS-architecture [9]. The existing QoS architecture on AIX supports policy-based outbound bandwidth management [10]. These techniques are easily portable to any OS running a BSD style network stack¹.

We present experimental results to demonstrate that these mechanisms effectively provide selective connection discard and service differentiation in an overloaded server. We also compare against application layer controls that we added in the Apache 1.3.12 server and show that the kernel controls are much more efficient and scalable.

The remainder of this paper is organized as follows: In Section 2 we give a brief overview on input packet processing. Our architecture and the kernel mechanisms are presented in Section 3. In Section 4 we present and discuss experimental results. We compare the performance of kernel based mechanisms and application level controls in Section 5. We describe related work in Section 6 and finally, the conclusions and future work in Section 7.

2 Input Packet Processing: Background

In this section we briefly describe the protocol processing steps executed when a new connection re-

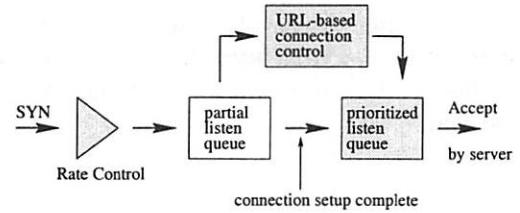


Figure 1: Proposed kernel mechanisms.

quest is processed by a web server. When the device interface receives a packet it triggers a hardware interrupt that is serviced by the corresponding device driver [11]. The device driver copies the received packet into an mbuf and de-multiplexes it to determine the queue to insert the packet. For example, an IP packet is added to the input queue, `ipintrq`. The device driver then triggers the IP software interrupt. The IP input routine dequeues the packet from the IP input queue and does the next layer demultiplexing to invoke the transport layer input routine. For example, for a TCP packet this will result in a call to a `tcp_input` routine for further processing. The call to the transport layer input routine happens within the realm of the IP input call, i.e., there is no queuing between the IP and TCP layer. The TCP input processing verifies the packet and locates the protocol control block (PCB). If the incoming packet is a SYN request for a listen socket, a new data socket is created and placed in the partial listen queue and an ACK is sent back to the client. When the ACK for the SYN-ACK is received the TCP 3-way handshake is complete, the connection moves to an established state and the data socket is moved to the listen queue. The sleeping process, e.g., the web server, waiting on the `accept` call is woken up. The connection is ready to receive data.

3 Architecture Design

The network subsystem architecture adds three control mechanisms that are placed at the different stages of a TCP connection's life time. Figure 1 shows the various phases in the connection setup and the corresponding control mechanisms: (i) when a SYN packet is processed it triggers the SYN rate control and selective drop (ii) when the 3-way handshake is completed the prioritized listen queue selectively changes the ordering of accepted connections in the listen queue (iii) when the HTTP header is received the HTTP header controls decide on dropping or re-prioritizing the requests based on application

¹A port to Linux is underway.

layer information. Each of these mechanisms can be activated at varying degrees of overload where the earliest and simplest control is triggered at the highest load level.

3.1 SYN Policer

TCP SYN policing controls the rate and burst at which new connections are accepted. Arriving TCP SYN packets are policed using a token bucket profile defined by the pair $\langle \text{rate}, \text{burst} \rangle$, where *rate* is the average number of new requests admitted per second and *burst* is the maximum number of concurrent new requests. Incoming connections are aggregated using specified filter rules that are based on the connection end points (source and destination addresses and ports as shown in Table 2). On arrival at the server, the SYN packet is classified using the IP/TCP header information to determine the matching rule. A compliance check is performed against the token bucket profile of the rule. If compliant, a new data socket is created and inserted in the partial listen queue otherwise the SYN packet is silently discarded.

When the SYN packet is silently dropped, the requesting client will time-out waiting for a SYN ACK and retry again with an exponentially increasing time-out value². An alternate option, which we do not consider, is to send a TCP RST to reset the connection indicating an abort from the server. This approach, however, incurs unnecessary extra overhead. Secondly, some clients send a new SYN immediately after a TCP RST is received instead of aborting the connection. Note that we drop non-compliant SYNs even *before* a socket is created for the new connection thereby investing only a small amount of overhead on requests that are dropped.

To provide service differentiation, connection requests are aggregated based on filters and each aggregate has a separate token bucket profile. Filtering based on client IP addresses is useful since a few domains account for a significant portion of a web server's requests [12]. The rate and burst values are enforced only when overload is detected and can be dynamically controlled by an adaptation agent, the details of which are beyond the scope of this paper.

²The timeout values are typically set to 6, 24, 48, up to 75 seconds.

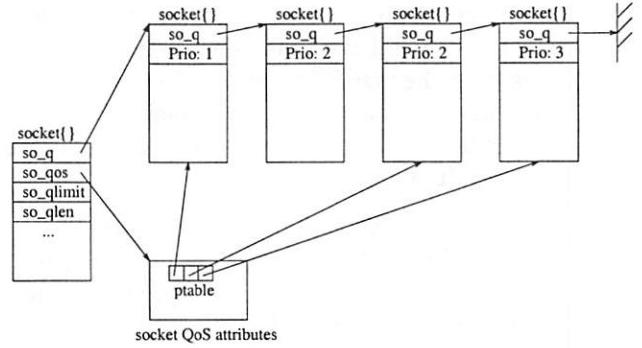


Figure 2: Implementation of the prioritized listen queue

3.2 Prioritized Listen Queue

The prioritized listen queue reorders the listen queue of a server process based on pre-defined connection priorities such that the highest priority connection is located at the head of the queue. The priorities are associated with filters (see Table 2) and connections are classified into different *priority classes*. When a TCP connection is established, it is moved from the partial listen queue to the listen queue. We insert the socket at the position corresponding to its priority in the listen queue. Since the server process always removes the head of the listen queue when calling *accept*, this approach provides better service, i.e. lower delay and higher throughput, to connections with higher priority.

Figure 2 shows the implementation of a prioritized listen queue. A special data structure used for maintaining socket QoS attributes stores an array of *priority pointers*. Each priority pointer points to the *last* socket of the corresponding priority class. This allows efficient socket insertion — a new socket is always inserted behind the one pointed to by the corresponding priority pointer.

3.3 HTTP Header-based Controls

The SYN policer and prioritized listen queue have limited knowledge about the type and nature of a connection request, since they are based on the information available in the TCP and IP headers. For web servers with the majority of the traffic being HTTP over TCP, a more informed control is possible by examining the HTTP headers. For example,

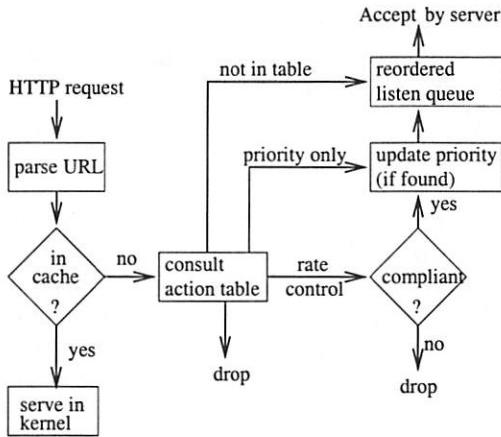


Figure 3: The HTTP header-based connection control mechanism.

Table 1: URL action table

URL	ACTION
noaccess	<drop>
/shop.html	<priority=1>
/index.html	< rate=15 conn./sec, burst=5 conn. >, <priority=1>
/cgi-bin/*	< rate=10 , burst=2 >

a majority of the load is caused by a few CGI requests and most of the bytes transferred belong to a small set of large files. This suggests that targeting specific URLs, types of URLs, or cookie information for service differentiation can have a wide impact during overload.

Our third mechanism, *HTTP header-based connection control*, enables content-based connection control by examining application layer information in the HTTP header, such as the URL name or type (e.g., CGI requests) and other application-specific information available in cookies. The control is applied in the form of rate policing and priorities based on URL names and types and cookie attributes.

This mechanism involves parsing the HTTP header in the kernel and waking the sleeping web server process only after a decision to service the connection is made. If a connection is discarded, a TCP RST is sent to the client and the socket receive buffer contents are flushed.

For URL parsing, our implementation relies upon Advanced Fast Path Architecture(AFPA) [13], an

Table 2: Example Network-level Policies

(dst IP,dst port,src IP,src port)	(r,b)	priority
(* , 80, *, *)	(300,5)	3
(* , 80, 10.1.1.1, *)	(100,5)	2
(12.1.1.1, 80, *, *)	(10,1)	*

in-kernel web cache on AIX. For Linux, an in-kernel web engine called KHTTPD is available [14]. As opposed to the normal operation, where the sleeping process is woken up after a connection is established, AFPA responds to cached HTTP requests directly without waking up the server process. With AFPA, a connection is *not* moved out of the partial listen queue even after the 3-way handshake is over. The normal data flow of TCP continues with the data being stored in the socket receive buffer. When the HTTP header is received (that is when the AFPA parser finds two CR control characters in the data stream), AFPA checks for the object in its cache. On a cache miss, the socket is moved to the listen queue and the web server process is woken up to service the request.

The HTTP header-based connection control mechanism comes into play at this juncture, as illustrated in Figure 3, before the socket is moved out of the partial listen queue. The URL action table (Table 1) specifies three types of actions/controls for each URL or set of URLs. A drop action implies that a TCP RST is sent before discarding the connection from the partial listen queue and flushing the socket receive buffer. If a priority value is set it determines the location of the corresponding socket in the ordered listen queue. Finally, rate control specifies a token bucket profile of a <rate, burst> pair which drops out-of-profile connections similar to the SYN policer.

3.4 Filter Specification

A filter rule specifies the network-level and/or application-level attributes that define an aggregate and the parameters for the control mechanism that is associated with it. A network-level filter is a four-tuple consisting of local IP address, local port, remote IP address, and remote port; application-level filters were shown in Table 1. Table 2 lists some network-level filter examples. The first rule applies to the web server process listening at local port 80 on all network interfaces; it specifies that all con-

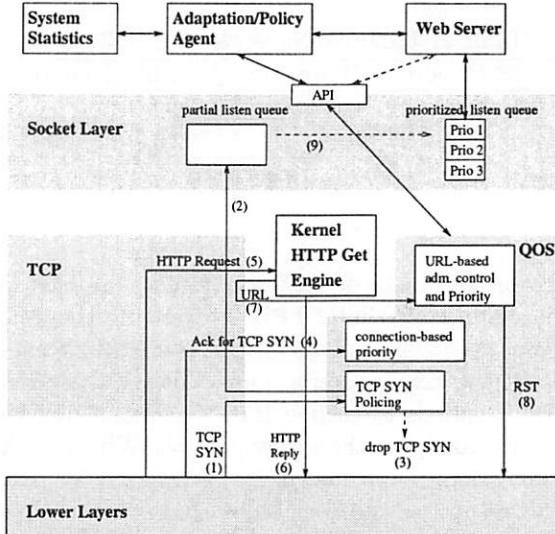


Figure 4: Enhanced protocol stack architecture.

nections to the server are rate-controlled at a rate of 300 conns/sec, a burst of 5, and a priority of 3 (the default lowest priority). The filter rules can contain range of IP addresses, wildcards, etc.

3.5 Protocol Stack Architecture

We have developed architectural enhancements for Unix-based servers to provide these mechanisms. Figure 4 shows the basic components of the enhanced protocol stack architecture, with the new capabilities utilized either by user-space agents or applications themselves. This architecture permits control over an application's inbound network traffic via policy-based traffic management [10]; an adaptation/policy agent installs policies into the kernel via a special API. The policy agent interacts with the kernel via an enhanced socket interface by sending (receiving) messages to (from) special control sockets. The policies specify filters to select the traffic to be controlled, and actions to perform on the selected traffic. The figure shows the flow of an incoming request through the various control mechanisms.

3.6 Implementation Methodology and Testbed

We have implemented the proposed kernel mechanisms in AIX 5.0, and evaluated them on the testbed

described below. As shown in Figure 4, the QoS module contains the TCP SYN policer, a priority assignment function for new connections, and the entity that performs URL-based admission control and priority assignment.

All experiments were conducted on a testbed comprising an IBM HTTP Server running on a 375 MHz RS/6000 machine with 512 MB memory, several 550 MHz Pentium III clients running Linux, and one 166 MHz Pentium Pro client running FreeBSD. The server and clients are connected via a 100 BaseT Ethernet switch. For client load generators we use Webstone 2.5 [15] and a slightly modified version of sclient [16]. Both programs measure client throughput in connections per second. The experimental workload consists of static and dynamic requests. The dynamic files are minor modifications of standard Webstone CGI files that simulate memory consumption of real-world CGIs.

The IBM HTTP Server is a modified Apache [17] 1.3.12 web server that utilizes an in-kernel HTTP get engine called the Advanced Fast Path Architecture (AFPA). We use AFPA in our architecture only to perform the URL parsing and have disabled any caching when measuring throughput results. Unless stated otherwise, we configured Apache to use a maximum of 150 server processes.

4 Experimental Results

4.1 Efficacy of SYN Policing

In this section we show how TCP SYN policing protects a preferred client against flash crowds or high request rates from other clients. In our setup, one client replays a large e-tailer's trace file representing a preferred customer. For the competing load we use five machines running Webstone, each with 50 clients. All clients request an 8 KB file, which is reasonable since a typical HTTP transfer is between 5 and 13 KB [12].

Without SYN policing, the e-tailer's client receives a low throughput of about 6 KB/sec. Using policing to lower the acceptance rate of Webstone clients, we expect the throughput for the e-tailer's client to increase. Figure 5 shows that the throughput for e-tailer's client increases from 100 KB/sec to 800 KB/sec as the acceptance rate for Webstone clients

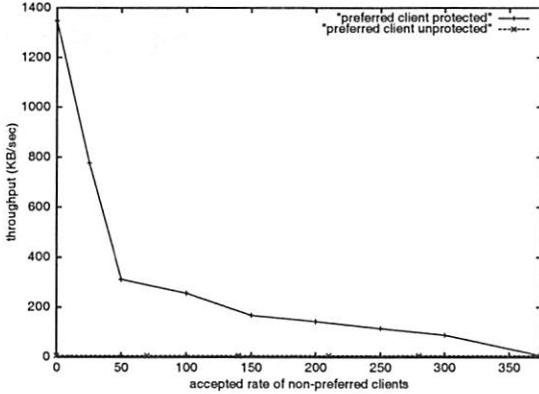


Figure 5: Throughput of the preferred e-tailer's client with and without TCP SYN policing. On the X-axis is the SYN policing rate of the non-preferred Webstone clients that are continuously generating requests. The Y-axis shows the corresponding throughput received by the e-tailer's client when there was no SYN control and when SYN control was enforced.

is lowered from 300 reqs/sec to 25 reqs/sec. The experiment demonstrates that a preferred client can be successfully protected by rate-controlling connection requests of other greedy clients.

TCP SYN policing works well when client identities and request patterns are known. In general, however, it is difficult to correctly identify a misbehaving group of clients. Moreover, as discussed below, it is hard to predict the rate control parameters that enable service differentiation for preferred clients without under-utilizing the server. For effective overload prevention the policing rate must be dynamically adapted to the resource consumption of accepted requests.

4.2 Impact of Burst Size

In the previous experiment we did not analyze the effect of the burst size on the effective throughput. The burst size is the maximum number of new connections accepted concurrently for a given aggregate. With a large burst size, greedy clients can overload the server, whereas with a small burst, clients may be rejected unnecessarily. The burst size also controls the responsiveness of rate control. There is a tradeoff, however, between responsiveness and the achieved throughput.

We next show the effect of the burst size on the

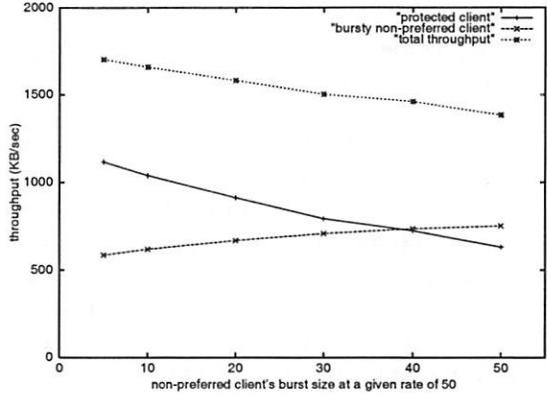


Figure 6: Impact of burst size on preferred client throughput. The burst size for policing non-preferred client is varied from 5 to 50 while the connection acceptance rate is fixed at 50 conn/sec. The plot shows the throughput achieved by the preferred and non-preferred clients along with the total throughput.

throughput of a preferred client. In our experiment, the non-preferred client is a modified sclient program that makes 50 to 80 back-to-back connection requests about twice a second, in addition to the specified request rate. Both the length of the incoming request burst and its timing are randomized. Figure 6 shows the throughput of preferred and non-preferred client with the SYN policing rate of the non-preferred client set to 50 conn/sec and the burst size varying from 5 to 50. The non-preferred sclient program requests a 16 KB dynamically generated cgi file. The preferred client is a Webstone program with 40 clients, requesting a static 8 KB file. As the burst size is increased from 5 to 50, the sclient's throughput increases from 36.6 conns/sec (585.6 KB/sec) to 47.7 conns/sec (752 KB/sec), while the throughput received by the preferred client decreases from about 140 conns/sec (1117 KB/sec) to 79 conns/sec.

Intuitively the overall throughput should have increased, however, the observed decrease in total throughput is due to the fact that we accept more CPU consuming CGI requests from sclient, thereby, incurring a higher overhead per byte transferred.

4.3 Prioritized Listen Queue: Simple Priority

With TCP SYN policing, one must limit the greedy non-preferred clients to a meaningful rate during overload. In most cases it is relatively simpler to just give the preferred clients a higher absolute pri-

ority. We demonstrate next that the prioritized listen queue provides service differentiation, especially with a large listen queue length.

In our experiments we classify clients into three priority levels. Clients belonging to a common priority level are all created by a Webstone benchmark that requests an 8 KB file. A separate Webstone instance is used for each priority level. We measure client throughput for each priority level while varying the total number of clients in each class. Each priority class uses the same number of clients.

In the first experiment, the Apache server is configured to spawn a maximum of 50 server processes. The results in Figure 7 show that when the total number of clients is small, all priority levels achieve similar throughput. With fewer clients, server processes are always free to handle incoming requests. Thus, the listen queue remains short and almost no reordering occurs. As the number of clients increases, the listen queue builds up since there are fewer Apache processes than concurrent client requests. Consequently, with re-ordering the throughput received by the high priority client increases, while that of the two lower priority clients decreases. Figure 7 shows that with more than 30 Webstone clients per class only the high-priority clients are served while the lower-priority clients receive almost no service.

Figure 8 illustrates the effect on response times observed by clients of the three priority classes. It can be seen that as the number of clients increases across all priority classes the response time for the lower priority classes increases exponentially. The response time of the high priority class, on the other hand, only increases sub-linearly. When the number of high priority requests increases, the lower priority ones are shifted back in the listen queue, thereby, increasing their response times. Also as more high priority requests get serviced by the different server processes running in parallel and competing for the CPU their response times increase.

We also observed that when the number of high priority requests was fixed and the lower priority request rate was steadily increased, the response time of the high priority requests remained unaffected.

The priority-based approach enables us to give low delay and high throughput to preferred clients independent of the requests or request patterns of

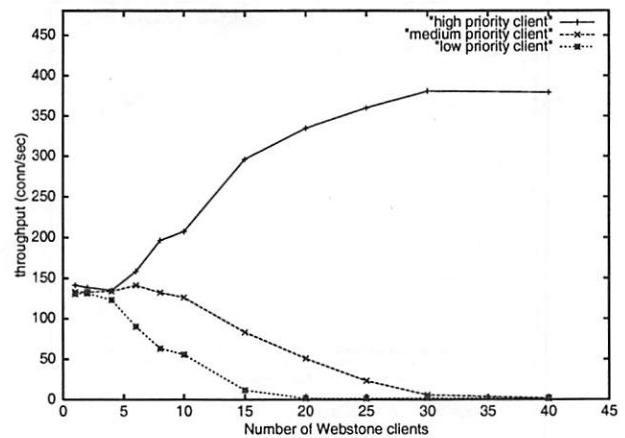


Figure 7: Throughput with the prioritized listen queue and 3 priority classes with 50 Apache processes. The number of clients in each class remains equal.

other clients. However, one may need many priority classes for different levels of service. The main drawback of a simple priority ordering is that it provides no protection against starvation of low-priority requests.

4.4 Combining Policing and Priority

To prevent starvation, low priority requests need to have some minimum number reserved slots in the listen queue so that they are not always preempted by a high priority request. However, reserving slots in the listen queue arbitrarily could cause a high priority request to find a full listen queue, which would in turn cause it to be aborted after its 3-way handshake is completed. To avoid starvation with fixed priorities, we combine the listen queue priorities with SYN policing to give preferred clients higher priority, but limiting their maximum rate and burst, thereby, implicitly reserving some slots in the queue for the lower priority requests.

Table 3 shows the results for experiments with three sets of Webstone clients with different priorities and rate control of the high priority class. The lower priority class has 30 Webstone clients while the high priority class has 150 Webstone clients spread over three different hosts. With no SYN policing of the clients in the high priority class, the two low-priority clients are completely starved. Table 3 shows that rate limiting the clients in the high priority class to 300 conn/sec prevents starvation; the medium and

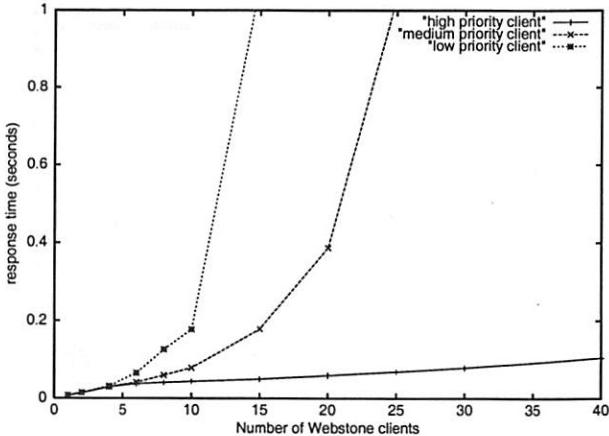


Figure 8: Response time with the prioritized listen queue and 3 priority classes with 50 Apache processes. The number of clients in each class remains equal.

Table 3: TCP SYN policing of a high-priority client to avoid starvation of other clients.

Throughput (conn/sec) of each priority class			
client priority	(rate, burst) limit of high priority		
	none	(300,300)	(200,200)
high	381	306	196
medium	0	78.6	180
low	0	4.1	13

low priority clients achieve a throughput of 78.6 and 4.1 conn/sec respectively.

4.5 HTTP Header-based Connection Control

In this section we illustrate the performance and effectiveness of admission control and service differentiation based on information in the HTTP headers i.e., URL name and type, cookie fields etc.

Rate control using URLs: In our experimental scenario the preferred client replaying the e-tailer's trace needs to be protected from overload due to a large number of high overhead CGI requests from non-preferred clients. The client issuing CGI requests is an sclient program requesting a dynamic file of length 5 KB at a very high rate. Figure 9 shows that without any protection, the preferred e-tailer's customer receives a low throughput of under 1 KB/sec. By rate-limiting the dynamic requests

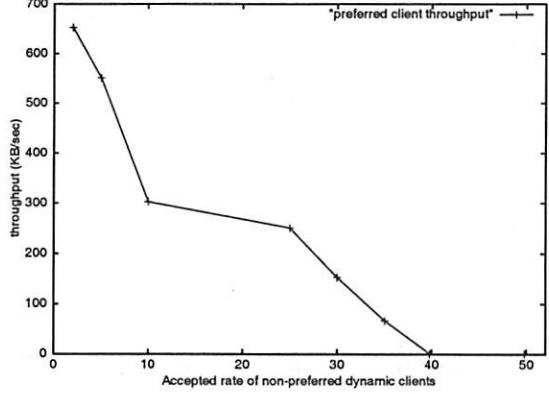


Figure 9: URL-based policing to protect preferred e-tailer's customers. The graph shows the resulting throughput of the preferred e-tailer's client as a specific high overhead CGI requests is limited to a given number of conn/sec

from 40 reqs/sec to 2 reqs/sec the throughput of the preferred e-tailer's customer improves from 1 KB/sec to 650 KB/sec. In contrast to TCP SYN policing (Figure 5), URL rate control targets a specific URL causing overload instead of a client pool.

URL priorities: In this section we present the results of priority assignments in the listen queue based on the URL name or type being requested. The clients are Webstone benchmarks requesting two different URLs, both corresponding to files of size 8 KB. There are two priority classes in the listen queue based on the two requested URLs. Figure 10 shows that the lower priority clients (accessing the low priority URL) receive lower throughput and are almost starved when the number of clients requesting the high priority URL exceeds 40. These results correspond to the results shown earlier with priorities based on the connection attributes (see Figure 7). The average total throughput, however, is slightly lower with URL-based priorities due to the additional overhead of URL parsing.

Combined URL-based rate control and priorities: To avoid starvation of requests for the low-priority URL, we rate limit the requests for the high-priority URL. In this experiment, we assign a higher priority to requests for a dynamic CGI request of size 5 KB (requested by an sclient program), and lower priority to requests for a static 8 KB file (requested by the Webstone program). Table 4 shows that starvation can be avoided by rate-limiting the high-priority URL requests.

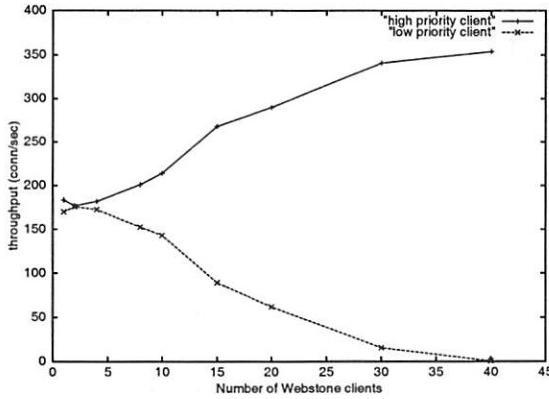


Figure 10: Throughput with 2 URL-based priorities and 50 Apache server processes. The number of clients in each class is equal

Table 4: URL-based policing of a high-priority client to avoid starvation of other clients.

Throughput (conn/sec)			
client priority	(rate, burst) limit of high priority		
	none	(30,10)	(10,10)
high	61.7	29.0	10.1
low	0	10.2	117

4.5.1 Overload Protection from High Overhead Requests

So far we have used the URL-based controls for providing service differentiation based on URL names and types. In the next experiment, we investigate if URL-based connection control can be used to protect a web server from overload by a targeted control of high overhead requests (e.g., CGI requests that require large computation or database access).

We use the sclient load generator to request a given high overhead URL and control the request rate, steadily increasing it and measuring the throughput. Figure 11 shows the client's throughput with varying request rates for a dynamic CGI request that generates a file size of 29 KB. The throughput increases linearly with the request rate up to a critical point of about 63 connections/sec. For any further increase in the request rate the throughput falls exponentially and later plateaus to around 40 connections/sec. To understand this behavior we used *vmstat* to capture the paging statistics. Since the dynamic requests are memory-intensive, the available free memory rapidly declines. For some combinations of the request rate

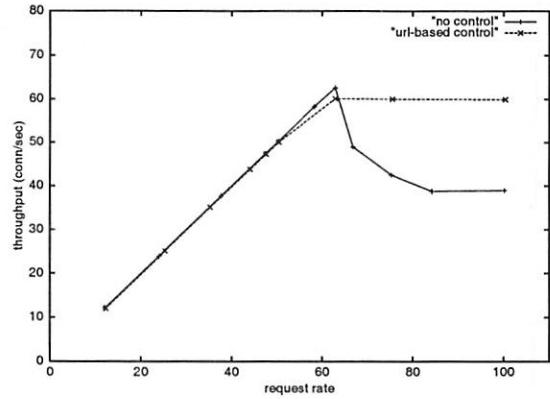


Figure 11: Overload protection from high overhead requests using URL-based connection control. The graph shows the throughput of web server with no controls servicing CPU intensive CGI requests and the corresponding throughput when the CGI requests are limited to 60 reqs/sec.

and the number of active processes, the available free memory falls to zero. Eventually the system starts thrashing as the CPU spends most of the time waiting for pending local disk I/O. In the above experiment with 150 server processes and a request rate of 63 reqs/sec the wait time starts increasing as indicated by the *wait* field of the output from *vmstat*.

To prevent overload we use URL-based connection control to limit the number of accepted dynamic CGI requests to a rate of 60 reqs/sec and a burst of 10. The dashed line in Figure 11 shows that with URL-based control the throughput stabilizes to 60 reqs/sec and the server never thrashes. In the above experiment, the URL-based connection control can handle request rates of up to 150 requests per second. However, for request rates beyond that thrashing starts as the kernel overhead of setting up connections, parsing the URL and sending the RSTs, becomes substantial.

To further delay the onset of thrashing we augment the URL-based control with the TCP SYN policer. For every TCP RST that is sent we drop any subsequent SYN request from that same client for a specified time interval. The time interval selected is the timeout value used for a lost SYN.

Table 5: Performance of AFPA and matching a URL to a rule for a 8 KB file with different URL lengths.

Throughput (conn/sec)			
URL off length	AFPA (no cache)	AFPA on, (no cache) no rule	AFPA on, matching rule
11 char.	370.1	340.5	338.3
80 char.	361.5	321.9	319.4
160 char.	355.1	321.1	303.7

Table 6: Overhead of kernel mechanisms

Operation		Cost(μsec)
TCP SYN policing	1 filter rule	7.9
	3 filter rules	9.6
classification and priority	1 rule	4.4
	3 rules	5.0
AFPA including URL parsing		19
URL-based rate control including URL matching	1 rule	5.0
	2 rules	5.8
	3 rules	6.5
URL-based priority including URL matching	1 rule	3.8
	2 rules	4.1
	3 rules	4.3

4.5.2 Discussion

The HTTP header-based rate control relies on sending TCP RST to terminate non-preferred connections as and when necessary. In a more user-friendly implementation we could directly return an HTTP error message (e.g., server busy) back to the client and close the connection.

Our current implementation of URL-based control handles only HTTP/1.0 connections. We are currently exploring different mechanisms for HTTP/1.1 with keep-alive connections to limit the number and types of requests that can be serviced on the same persistent connection. The experiments in the previous section have only presented results on URL based controls. Similar controls can be set based on the information in cookies that can capture transaction information and client identities.

4.6 Overhead of the Kernel Mechanisms

We quantify the overhead of matching URLs in the kernel for varying URL lengths. Table 5 shows that the overhead of matching a URL to a rule is moderate (under 6% for a 160 character URL). The throughput numbers are for 20 Webstone clients requesting an 8 KB file. Rules are matched using the standard string comparison (`strcmp`) with no optimizations; better matching techniques can reduce this overhead significantly. On a cache miss, the in-kernel AFPA cache introduces an overhead of about 10% for an 8 KB file. However, the AFPA cache under normal conditions increases performance significantly for cache hits. In our experiments we have the cache size set to 0 so that AFPA cannot serve any object from the cache. When caching is enabled Webstone received a throughput of over 800 connections per second on a cache hit.

Table 6 summarizes the additional overhead of the implemented kernel mechanisms. The overhead of compliance check and filter matching for TCP SYN policing with 1 filter rule is 7.9 μsecs. Simply matching the filter, allocating space to store QoS state, and setting the priority adds an overhead of around 4.4 μsecs for 1 filter rule. The policing controls are more expensive as they include accessing the clock for the current time. Surprisingly, the URL matching and rate control has a low overhead of 5.0 μsecs for a URL of 11 chars. This happens to be lower than SYN policing as the `strcmp` matching is cheaper for one short URL compared to matching multiple IP addresses and port numbers. The overhead of URL matching and setting priorities for a single rule is around 3.8 μsecs. The most expensive operation is the call to AFPA to parse the URL. AFPA not only parses the URL, but also does logging, checks if the requested object is in the network buffer cache, and pre-computes the HTTP response header.

5 Comparison of User Space and Kernel Mechanisms

In this section we compare the effectiveness of our kernel mechanisms with overload protection and service differentiation mechanisms implemented in user space. One might argue that kernel-based mechanisms are less flexible and more difficult to implement than mechanisms implemented in user space. User level controls although limited in their capa-

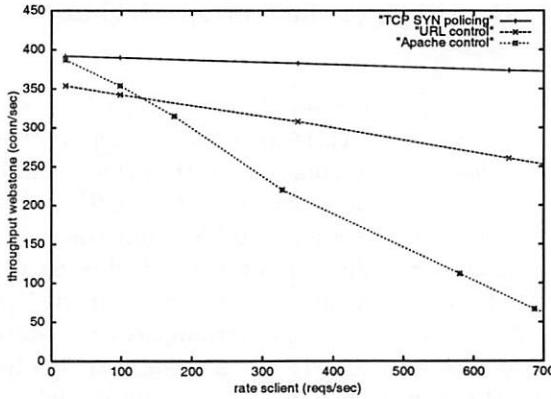


Figure 12: Throughput of kernel-based TCP SYN policing, kernel based URL rate control and Apache module based connection rate control. The throughput achieved by Webstone clients is measured against an increasing request load generated by sclient. The sclient requests are rate controlled to 10.0 req/sec with a burst of 2.

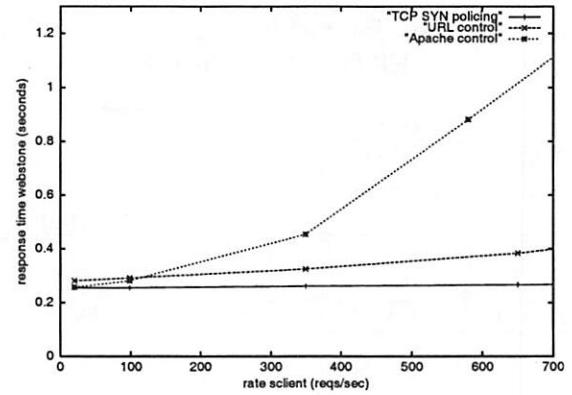


Figure 13: Response times using kernel-based TCP SYN policing, kernel based URL rate control and Apache module based connection rate control. The response time achieved by Webstone clients is measured against an increasing request load generated by sclient. The sclient requests are rate controlled to 10 req/sec with a burst of 2.

bilities, have easy access to application layer information. However, kernel mechanisms are more scalable and provide much better performance. In general, placing mechanisms in the kernel is beneficial if it leads to considerable performance gains and increases the robustness of the server without relying on the application layer to prevent overload.

To enable a fair comparison we have extended the Apache 1.3.12 server with additional modules [18] that police requests based on the client IP address and requested URL. The implemented rate control schemes use exactly the same algorithms as our kernel based mechanisms. If a request is not compliant we send a “server temporarily unavailable” (503 response code) back to the client and close the connection.

The experimental setup consists of a Webstone traffic generator with 100 clients requesting a file of size 8 KB along with an sclient program requesting a file of size 16 KB. The sclient’s requests are rate controlled with a rate of 10 requests per second and a burst of 2; there are no controls set for the Webstone clients. During our experiments, we steadily increased the sclient’s request rate.

Figure 12 illustrates that when the request load of the sclient program is low (20 reqs/sec), the Webstone throughput is 392 conn/sec and 387.3 conn/sec for TCP SYN policing and Apache user level controls respectively. These controls limit the sclient acceptance rate to 10.0 conn/sec. With in-kernel

URL-based rate control the throughput is lower (354 conn/sec). This low throughput is caused by the additional 10% overhead added by AFPA (with no caching) as discussed in Section 4.6. As discussed earlier, with the cache size set to zero, we add more overhead than necessary for URL parsing, without the corresponding gains from AFPA caching.

As the sclient’s request load increases further, TCP SYN policing is able to achieve a sustained throughput for the Webstone clients, while the Apache based controls shows a marked decline in throughput. The graph shows that for a sclient load of 650 reqs/sec the Webstone throughput for TCP SYN policing is 374 conn/sec; for in-kernel URL-based connection control it is 260.7 conn/sec; for Apache user level controls the throughput sinks to about 75 conn/sec. The corresponding results for response times are shown in Figure 13.

The experiment demonstrates that the kernel mechanisms are more efficient and scalable than the user space mechanisms. There are two main reasons for the higher efficiency and scalability: First, non-compliant connection requests are discarded earlier reducing the queuing time of the compliant requests, in particular less CPU is consumed and the context switch to user space is avoided. Second, when implementing rate control at user space, the synchronization mechanisms for sharing state among all the Apache server processes decrease performance.

6 Related Work

Several research efforts have focused on admission control and service differentiation in web servers [19], [20], [21], [22], [8] and [23]. Almeida *et al.* [8] use priority-based schemes to provide differentiated levels of service to clients depending on the web pages accessed. While in their approach the application, i.e., the web server, determines request priorities, our mechanisms reside in the kernel and can be applied without context-switching to user level. *WebQoS* [23] is a middleware layer that provides service differentiation and admission control. Since it is deployed in user space, it is less efficient compared to kernel-based mechanisms. While WebQoS also provides URL-based classification, the authors do not present any experiments or performance considerations. Cherkasova *et al.* [20] present an enhanced web server that provides session-based admission control to ensure that longer sessions are completed. Crovella *et al.* [24] show that client response time improves when web servers serving static files serve shorter connections before handling longer connections. Our mechanisms are general and can easily realize such a policy.

Reumann *et al.* [25] have presented virtual services, a new operating system abstraction that provides resource partitioning and management. Virtual services can enhance our scheme by, for example, dynamically controlling the number of processes a web server is allowed to fork. In [26] Reumann *et al.* have described an adaptive mechanism to setup rate controls for overload protection. The receiver livelock study [2] showed that network interrupt handling could cause server livelocks and should be taken into consideration when designing process scheduling mechanisms. Banga and Druschel's [27] *resource containers* enable the operating system to account for and control the consumption of resources. To shield preferred clients from malicious or greedy clients one can assign them to different containers. In the same paper they also describe a multi listen socket approach for priorities in which a filter splits a single listen queue into multiple queues from which connections are accepted separately and accounted to different principals. Our approach is similar, however, connections are accepted from the same single listen queue but inserted in the queue based on priority. Kanodia *et al.* [21] present a simulation study of queuing-based algorithms for admission control and service differentiation at the front-end. They focus

on guaranteeing latency bounds to classes by controlling the admission rate per class. Aron *et al.* [28] describe a scalable request distribution architecture for clusters and also present resource management techniques for clusters.

Scout [29], Rialto [30] and Nemesis[31] are operating systems that track per-application resource consumption and restrict the resources granted to each application. These operating systems can thus provide isolation between applications as well as service differentiation between clients. However, there is a significant amount of work involved to port applications to these specialized operating systems. Our focus, however, was not to build a new operating system or networking architecture but to introduce simple controls in the existing architecture of commercial operating systems that could be just as effective.

7 Conclusions and Future Work

In this paper, we have presented three in-kernel mechanisms that provide service differentiation and admission control for overloaded web servers. TCP SYN policing limits the number of incoming connection requests using a token bucket policer and prevents overload by enforcing a maximum acceptance rate of non-preferred clients. The prioritized listen queue provides low delay and high throughput to clients with high priority, but can starve low priority clients. We show that starvation can be avoided by combining priorities with TCP SYN policing. Finally, URL-based connection control provides in-kernel admission control and priority based on application-level information such as URLs and cookies. This mechanism is very powerful and can, for example, prevent overload caused by dynamic requests. We compared the kernel mechanisms to similar application layer controls added in the Apache server and demonstrated that the kernel mechanisms are much more efficient and scalable than the Apache user level controls.

The kernel mechanisms that we presented rely on the existence of accurate policies that control the operating range of the server. In a production system it is unrealistic to assume knowledge of the optimal operating region of the server. We are currently implementing a policy adaptation agent (Figure 4) that dynamically adapts the rate control policies to the changing workload conditions. This adaptation

agent uses available kernel statistics and past history to select appropriate values for the various policies and monitors the interaction between various control options on the overall performance during overload.

Our current implementation does not address security issues of fake IP addresses and client identities. We plan to integrate a variety of overload prevention policies with traditional firewall rules to provide an integrated solution.

References

- [1] "Cisco TCP intercept," <http://www.cisco.com>.
- [2] J. C. Mogul and K. K. Ramakrishnan, "Eliminating receive livelock in an interrupt-driven kernel," in *Proc. of USENIX Annual Technical Conference*, Jan. 1996.
- [3] P. Druschel and G. Banga, "Lazy receiver processing (LRP): a network subsystem architecture for server systems," in *Proc. of OSDI*, Oct. 1996, pp. 91-105.
- [4] O. Spatscheck and L. Peterson, "Defending against denial of service attacks in scout," in *Proc. of OSDI*, Feb. 1999.
- [5] "Ensim corporation: virtual servers," <http://www.ensim.com>.
- [6] "Alteon web systems," <http://www.alteonwebsystems.com>.
- [7] "Cisco arrowpoint web network services," <http://www.arrowpoint.com>.
- [8] J. Almeida, M. Dabu, A. Manikutty, and P. Cao, "Providing differentiated levels of service in web content hosting," in *Proc. of Internet Server Performance Workshop*, Mar. 1999.
- [9] T. Barzilai, D. Kandlur, A. Mehra, and D. Saha, "Design and implementation of an rsvp based quality of service architecture for an integrated services internet," *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 3, pp. 397-413, Apr. 1998.
- [10] A. Mehra, R. Tewari, and D. Kandlur, "Design considerations for rate control of aggregated tcp connections," in *Proc. of NOSSDAV*, June 1999.
- [11] G.R. Wright and W.R. Stevens, *TCP/IP Illustrated, Volume 2*, Addison-Wesley Publishing Company, 1995.
- [12] Martin F. Arlitt and Carey I. Williamson, "Web server workload characterization: The search for invariants," in *Proc. of ACM Sigmetrics*, Apr. 1996.
- [13] P.Joubert, R. King, R.Neves, M.Russinovich, and J.Tracey, "High performance memory based web caches: Kernel and user space performance," in preparation.
- [14] "Khttpd - linux http accelerator," <http://www.fenrus.demon.nl/>.
- [15] "webstone," <http://www.mindcraft.com>.
- [16] G. Banga and P. Druschel, "Measuring the capacity of a web server," in *Proc. of USITS*, Dec. 1997.
- [17] "apache," <http://www.apache.org>.
- [18] L. Stein and D. MacEachern, *Writing Apache modules with Perl and C*, O'Reilly, 1999.
- [19] T. Abdelzaher and N. Bhatti, "Web server qos management by adaptive content delivery," in *Intl. Workshop on Quality of Service*, June 1999.
- [20] L. Cherkasova and P. Phaal, "Session based admission control: a mechanism for improving the performance of an overloaded web server," Tech. Rep., Hewlett Packard, 1999.
- [21] V. Kanodia and E. Knightly, "Multi-class latency-bounded web servers," in *Intl. Workshop on Quality of Service*, June 2000.
- [22] K. Li and S. Jamin, "A measurement-based admission controlled web server," in *Proc. of INFOCOMM*, Mar. 2000.
- [23] Nina Bhatti and Rich Friedrich, "Web server support for tiered services," *IEEE Network*, Sept. 1999.
- [24] M. E. Crovella, R. Frangioso, and M. Harchol-Balter, "Connection scheduling in web servers," in *Proc. of USITS*, Oct. 1999.
- [25] J. Reumann, A. Mehra, K. Shin, and D. Kandlur, "Virtual services: A new abstraction for server consolidation," in *Proc. of USENIX Annual Technical Conference*, June 2000.
- [26] H. Jamjoom and J. Reumann, "Qguard:protecting internet servers from overload," Tech. Rep., University of Michigan, CSE-TR-427-00, 2000.
- [27] G. Banga, P. Druschel, and J. Mogul, "Resource containers: a new facility for resource management in server systems," in *Proc. of OSDI*, Feb. 1999.
- [28] M. Aron, D. Sanders, P. Druschel, and W. Zwaenepoel, "Scalable content-aware request distribution in cluster-based network servers," in *Proc. of USENIX Annual Technical Conference*, June 2000.
- [29] D. Mosberger and L. L. Peterson, "Making paths explicit in the scout operating system," in *Proc. of OSDI*, Oct. 1996, pp. 153-167.
- [30] M. B. Jones, J. S. Barrera III, A. Forin, P. J. Leach, D. Rosu, and M. Rosu, "An overview of the Rialto real-time architecture," in *ACM SIGOPS European Workshop*, Sept. 1996, pp. 249-256.
- [31] Thiendo Voigt and Bengt Ahlgren, "Scheduling TCP in the Nemesis operating system," in *IFIP WG 6.1/WG 6.4 International Workshop on Protocols for High-Speed Networks*, Aug. 1999.

Storage management for web proxies

Elizabeth Shriver

Bell Laboratories

shriver@bell-labs.com

Eran Gabber

Bell Laboratories

eran@bell-labs.com

Lan Huang

SUNY Stony Brook

lanhuang@cs.sunysb.edu

Christopher A. Stein

Harvard University

stein@eecs.harvard.edu

Abstract

Today, caching web proxies use general-purpose file systems to store web objects. Proxies, e.g., Squid or Apache, when running on a UNIX system, typically use the standard UNIX file system (UFS) for this purpose. UFS was designed for research and engineering environments, which have different characteristics from that of a caching web proxy. Some of the differences are high temporal locality, relaxed persistence requirements, and a different read/write ratio. In this paper, we characterize the web proxy workload, describe the design of Hummingbird, a light-weight file system for web proxies, and present performance measurements of Hummingbird. Hummingbird has two distinguishing features: it separates object naming and storage locality through direct application-provided hints, and its clients are compiled with a linked library interface for memory sharing. When we simulated the Squid proxy, Hummingbird achieves document request throughput 2.3–9.4 times larger than with several different versions of UFS. Our experimental results are verified within the Polygraph proxy benchmarking environment.

1 Introduction

Caching web proxies are computer systems dedicated to caching and delivering web content. Typically, they exist on a corporate firewall or at the point where an Internet Service Provider (ISP) peers with its network access provider. From a web performance and scalability point of view, these systems have three purposes: improve web client latency, drive down the ISP's network access costs because of reduced bandwidth requirements, and reduce request load on origin servers.

Squid and Apache are two popular web proxies. Both of these systems use the standard file system services provided by the host operating system. On UNIX this is usually UFS, a descendant of the 4.2BSD UNIX Fast File System (FFS) [13]. FFS was designed for workstation workloads and is not optimized for the different workload and requirements of a web proxy. It has been observed that file system latency is a key component in the latency observed by web clients [21].

Some commercial vendors have improved I/O performance by rebuilding the entire system stack: a special operating system with an application-specific file system executing on dedicated hardware (e.g., CacheFlow, Network Appliance). Needless to say, these solutions are expensive. We believe that a lightweight and portable file system can be built that will allow proxies to achieve performance close to that of a specialized system on commodity hardware, within a general-purpose operating system, and with minimal changes to their source code; Gabber and Shriver [5] discuss this view in detail.

We have built a simple, lightweight file system library named Hummingbird that runs on top of a raw disk partition. This system is easily portable—we have run it with minimal changes on FreeBSD, IRIX, Solaris, and Linux. In this paper we describe the design, interface, and implementation of this system along with some experimental results that compare the performance of our system with a UNIX file system. Our results indicate that Hummingbird's throughput is 2.3–4.0 times larger than a simulated version of Squid running UFS mounted asynchronously on FreeBSD, 5.4–9.4 times faster than Squid running UFS mounted synchronously on FreeBSD, 5.6–8.4 times larger than a simulated version of Squid running UFS with soft updates on FreeBSD, and 5.4–13 times larger than XFS and

EFS on IRIX (see Section 4). We also performed experiments using the Polygraph environment [18] with an Apache proxy; the mean response time for hits in the proxy is 14 times smaller with Hummingbird than with UFS (see Section 5).

Throughout the rest of this paper, we use the terms *proxy* or *web proxy* to mean *caching web proxy*. Section 2 presents the important characteristics of the proxy workload considered for our file system. It also presents some background on file systems and proxies that is important because it motivates much of our design. Section 3 describes the Hummingbird file system. Our experiments and results are presented in Sections 4 and 5. Section 6 discusses related work in file systems and web proxy caching.

2 File system limitations

The web proxy workload is different than the standard UNIX workload; these differences are discussed in Section 2.1. Due to these differences, the performance which can be attained for a web proxy running on UFS is limited by some of the features; these limitations are discussed in Section 2.2.

2.1 Web proxy workload characteristics

Web proxy workloads have special characteristics, which are different from those of a traditional UNIX file system workload. This section describes the special characteristics of proxy workloads.

We studied a week's worth of web proxy logs from a major, national ISP, collected from January 30 to February 5, 1999. This proxy ran Netscape Enterprise server proxy software and the logs were generated in Netscape-Extended2 format. For the purpose of our analysis we isolated the request stream to those that would affect the file system underlying the proxy. Thus we excluded 34% of the GET requests which are considered non-cacheable by the proxy. If we could not find the file size, we removed the log event. This preprocessing results in the removal of about 4% of the log records, nearly all during the first few days. We eliminated the first few days and are left with 4 days of processed logs containing 4.8 million requests for 14.3 GB of unique cacheable data and 27.6 GB total requested cacheable data.

Characteristics of a web proxy and its workload are:

Persistence. A web proxy only writes files retrieved from an origin server. Thus, these are just cached files, and can be retrieved again if necessary.

Naming. The web proxy application determines the name of a file to be written into the file system. In a traditional UNIX file system, file names are normally selected by a user.

Reference locality. Client web accesses are characterized by a request for an HTML page followed by requests for embedded images. The set of images within a cacheable HTML page changes slowly over time. So, if a page is requested by a client, it makes sense for the proxy to anticipate future requests by prefetching its historically associated images.

We studied one day of the web proxy log for this reference locality. The first time we see an HTML file, we coalesce it with all following non-HTMLs from the same client to form the primordial *locality set*, which does not change. The next time the HTML file is referenced, we form another locality set in the same manner and compare its file members with those of the primordial locality set. The average hit rate (the ratio between the size of the latter sets and the size of the primordial set) across all references is 47%. Thus, on average, a locality set re-reference accesses almost half of the files of the original reference. One of the reasons that this hit rate is small might be due to the assumption that all non-HTML files that follow a HTML file are in the same locality set; this is clearly not true if a user has multiple active browser sessions. Also, we determined the type of file using the file extension, thus possibly placing some HTML files in another file's locality set. We also studied the size of the locality sets in bytes, and found that 42% are 32 KB or smaller, 62% are 64 KB or smaller, 73% are 96 KB or smaller, and 80% are 128 KB or smaller.

File access. Several older UNIX file system performance studies have shown that files are usually accessed sequentially [16, 1]. A recent study [19] has suggested that this is changing, especially for memory-mapped and large files. However, UNIX file systems have been designed for the traditional sequential workload. Web proxies have an even stronger and more predictable pattern of behavior. Web proxies currently always access files sequentially and in their entirety. (This may change when range requests are more popular.)

File size. Most cacheable web documents are small; the median size of requested files is 1986 B and the average size is 6167 B. Over 90% of the references are for files smaller than 8 KB.

Idleness. The proxy workload is characterized by a large variability in request rate and frequent idle

periods. Using simulation, we found, in fact, in each 1-minute interval, the disk is idle 57–99% of the duration of the interval, with a median of 82%. If the request rate in the trace doubles, the disk is idle less of the time, with a median of 60%. In the busiest periods, it is idle only 1%. Note that when the request rate doubles, the disk is almost saturated in the busy intervals (idle time drops to 1%), while the disk remains idle in the low-activity intervals. This idleness study used assumptions that represent Hummingbird: the disk has 64 KB blocks, all new files are written to the disk, and only data needs to be written to disk (no meta-information). The existence of idle periods is crucial for the design of Hummingbird since Hummingbird performs background bookkeeping operations in those idle periods (see Section 3.5).

2.2 UFS performance limitations

Due to the proxy workload characteristics presented in the previous section, UFS has a number of features which are not needed, or could be greatly simplified, so that file system performance could be improved. Table 1 presents the features on which UFS and a desired caching web proxy file system should differ. We now discuss these features.

UFS files are collections of fixed-size blocks, typically 8 KB in size. When accessing a file, the disk delays are due to disk head positioning occurring when the file blocks are not stored contiguously on disk. UFS attempts to minimize the disk head positioning time by storing the file blocks contiguously and prefetching blocks when a file is accessed sequentially, and does a good job of this for small files. Thus, when the workload consists of mostly small files, the largest component of disk delays are due to the reference stream locality not corresponding with the on-disk layout. UFS attempts to reduce this delay by having the user place files into directories, and locates files in a directory on a group of contiguous disk blocks called cylinder groups. Thus, reference locality is tied to naming. Here, the responsibility for performance lies with the application or user, who must construct a hierarchy with directory locality that matches future usage patterns. In addition, to reduce file lookup times, the directory hierarchy must be well-balanced and any single directory should not have too many entries. Squid attempts to balance the directory hierarchy, but in the process distributes the reference stream across directories, thus destroying locality. Apache maps files from the same origin server into the same directory. For specifying locality by a web proxy, a more

direct and low-overhead mechanism can be used.

Experience with Squid and Apache has shown that it is difficult for web proxies to use directory hierarchies to their advantage [11]. Deep pathnames mean long traversal times. Populated directories slow down lookup further because many legacy implementations still do a linear search for the file name through the directory contents. The hierarchical name space allows files to be organized by separating them across directories, but this is not needed by a proxy. What the proxy actually needs is a flat name space and the ability to specify storage locality.

UFS file meta-data is stored in the i-node, which is updated using synchronous disk writes to ensure meta-data integrity. A caching web proxy does not require file durability for correctness, so it is free to replace synchronous meta-data writes with asynchronous writes to improve the performance (which is done with soft updates [6]) and eliminate the need to maintain much of the meta-data associated with persistence.

Traditional file systems also force two architectural issues. First, the standard file system interface copies from kernel VM into the applications' address space. Second, the file system caches file blocks in its own buffer cache. Web proxies manage their own application-level VM caches to eliminate memory copies and use private information to facilitate more effective cache management. However, web documents cached at the application level are likely to also exist in the file system buffer cache, especially if recently accessed from disk. This multiple buffering reduces the effective size of the memory. A single unified cache solves the multiple buffering and configuration problems. Memory copy costs can be alleviated by passing data by reference rather than copying. Both of these can be done using a file system implemented by a library that accesses the raw disk partition. Another approach for alleviating multiple buffering is using memory-mapped files as done by Maltzahn et al. [11].

3 File system design

The design of Hummingbird is influenced by the proxy workload characteristics discussed in Section 2. Hummingbird uses locality hints generated by the proxy to pack files into large, fixed-size extents called *clusters*, which are the unit of disk access. Therefore, reads and writes are large, amortized

Table 1: Comparison of file system features.

feature	UFS	Hummingbird
name space	hierarchical name space	flat name space
reference locality	application manages directories	application sends locality hints
file meta-data	meta-data kept on disk; synchronous updates preserve consistency	most meta-data in memory
disk layout of files	data in blocks, with separate i-node for meta-data	file and meta-data stored contiguously
disk access	could be as small as a block size	cluster size (typically 32 or 64 KB)
interface	buffers are passed; memory copies are necessary	pointers are passed

ing disk positioning times and interrupt processing.

Hummingbird manages a large memory cache. Since Hummingbird is implemented by a library that is linked in with the proxy, no memory copies or memory mappings are required to move data from the file system to the proxy. The file system simply passes a pointer. Likewise, the proxy passes the file system a pointer when it writes data. We have only designed the file system for a single client, so protection is not necessary. Since the client (the proxy) handles all data transfers to and from the system, it must be trusted in any case. The proxy may be multi-threaded or have multiple processes¹, in which case access is serialized with a single lock on the file system meta-data that is released before blocking for disk I/O. Using a single lock may slow the system under a heavy load. Since the file system is I/O bound, the lock is held only when referring to Hummingbird's data structures. The lock is released before a thread blocks for disk I/O. No locking is needed when accessing file data.

Since the typical workload is bursty, Hummingbird is designed to reduce the response time during bursty periods, and perform maintenance activities during the idle periods present in the workload. Hummingbird performs the maintenance activities by calling several daemons responsible for: (1) reclaiming main memory space by writing files into clusters, and (2) reclaiming disk space by deleting unused clusters.

While there are invariants across proxy workloads, some characteristics will change. We have designed Hummingbird to be configurable so that the system can be optimized for a proxy workload and the underlying storage hardware. To this effect, Hum-

mingbird has several parameters that the proxy is free to set to optimize the system for its workload. The parameters set at file system initialization include: size of a cluster, memory cache eviction policy, file hash table size, file and cluster lifetimes, disk data layout policy, and recovery policies.

3.1 Hummingbird objects

Hummingbird stores two main types of objects in main memory: *files* and *clusters*. A file is created by a `write_file()` call. Clusters contain files and some file meta-data. Grouping files into clusters allows the file system to physically collocate files together, since when a cluster is read from disk, all of the files contained in the cluster are read. Clusters are *clean*, i.e., they can be evicted from main memory by reclaiming their space without writing to disk, since a cluster is written to disk as soon as it is created. (Section 3.5 discusses where on disk the clusters are written.)

The application provides locality hints by the `collocate_files(fnameA, fnameB)` call. The file system saves these hints until the files are assigned to clusters. This assignment occurs as late as possible, that is, when space is needed in main memory. At this point, the file system attempts to write `fnameA` and `fnameB` in the same cluster. It is possible for a file to be a member of multiple clusters, and stored in multiple locations on disk by the application sending multiple hints (e.g., `collocate_files(fnameA, fnameB)` and `collocate_files(fnameC, fnameB)`). For proxy caches, this is a useful feature since embedded images are frequently referenced in a number of related HTML pages.

When the file system is building a cluster, it determines which files to add to the cluster using an LRU ordering according to the last time the file was read. If the least-recently-used file has a list of collocated

¹To perform memory management for the multi-process version of Hummingbird, the processes have a shared memory region which is used for `malloc()`.

files, then these files are added to the cluster if they are in main memory. (If a file is on the collocation list, and already has been added to a cluster, it can still be added to the current cluster if the file is in memory.) Files are packed into the cluster until the cluster size threshold is reached, or until all files on the LRU list have been processed. This way, small locality sets with similar last-read times can be packed into the same cluster. Another possible algorithm to pack files into clusters is the Greedy Dual Size algorithm [3].

Large files are special. They account for a very small fraction of the requests, but a significant fraction of the bytes transferred. In the log we analyzed, files over 1 MB accounted for over 8% of the bytes transferred, but only 0.02% of the requests. Caching these large files is not important for the average latency perceived by clients, but is an important factor in the network access costs of the ISP. It is better to store these large files on disk, and not in the file system cache in memory. The `write_nOMEM_file()` call bypasses the main memory cache and writes a file directly to disk; if the file is larger than the cluster size, multiple clusters are allocated. Having an explicit `write_nOMEM_file()` function allows the application to request that any file can bypass main memory, not just large files.

3.2 Meta-data

Hummingbird maintains three types of meta-data: file system meta-data, file meta-data, and cluster meta-data.

File system meta-data. To determine when main memory space needs to be freed, Hummingbird maintains counts of the amount of space used for storing the files and the file system data. To assist with determining which files and clusters to evict from main memory, Hummingbird maintains two LRU lists, one for files which have not yet been packed into clusters and another for clusters that are in memory.

File meta-data. A hash table stores pointers to the file information such as the file number (discussed below), status, and a reference count of the number of users that are currently reading the file. The file status field identifies whether the file is not in a cluster, in one cluster, in multiple clusters, or not cacheable. Until a file becomes a member of a cluster, the file name and file size need to be maintained as part of the file meta-data. We also maintain a list of files that should be collocated with this file. When a file is added to a cluster, the file

meta-data must include the cluster ID and the file reference count for that file.

It is natural for a proxy to use the URL as a file name. URLs may be extremely long, and since we have many small files, the file names may take up a large portion of main memory if they were kept permanently in memory. Thus, we save the file name with the file data in its cluster and not permanently in memory. Internally, Hummingbird hashes the file name into a 32-bit index, which is used to locate the file meta-data. Hash collisions can be detected by comparing the requested file name with the file name stored in the cluster. If there is a collision, the next element in the hash table bucket is checked.

Cluster meta-data. A cluster table contains information about each cluster on disk: the status, last-time accessed, and a linked list of the files in the cluster. The cluster status field identifies whether the cluster is empty, on disk, or in memory. For our file system, the cluster ID identifies the location of the cluster on disk. While a cluster is in memory, the address of the cluster in memory is needed. The last-time accessed is needed to determine the amount of time since the cluster was last touched.

3.3 The Hummingbird interface

This section describes the basic Hummingbird calls. All routines return a positive integer or zero if the operation succeeds; a negative return value is an error code.

- `int write_file(char* fname, void* buf, size_t sz);` This function writes the contents of the memory area starting at `buf` with size `sz` to the file system with filename `fname`. It returns the size of the file. Once the pointer `buf` is handed over to the file system, the application **should not** use it again. Hummingbird will eventually pack the file into a cluster, free the buffer, and write the cluster to stable storage.
- `int read_file(char* fname, void** buf);` This function sets `*buf` to the beginning of the memory area containing the contents of the file `fname`. It increments a reference count and returns the size of the file. If the file is not in main memory, another file or files may be evicted to make room, and a cluster containing the specified file will be read from disk.
- `int done_read_file(char* fname, void* buf);` This function releases the space occupied by the file in main memory by decrementing the reference count; the application **should not** use the pointer again. Every `read_file()` must be accompanied

by a `done_read_file()`. Otherwise, the file will stay in memory indefinitely (until the application program terminates).

- `int delete_file(char* fname);` This function deletes the file `fname`. An error code is returned if the file has any active `read_file()`'s.
- `int collocate_files(char* fnameA, char* fnameB);` This function attempts to collocate file `fnameB` with file `fnameA` on disk. Both files must be previously written (by calling `write_file()`).
- `int write_nomem_file(char* fname, void* buf, size_t sz);` This function bypasses the main memory cache and writes a file directly to disk. This file is flagged so that when it is read, it does not compete with other documents for cache space and is immediately released after the application issues the `done_read_file()`.

Missing from this API are commands such as `ls` and `df`. We have seen no need for such commands for a caching web proxy. For example, the existence of a file can be determined with `read_file()`.

3.4 Recovery

Web proxies are slowly convergent; it takes days to reach the maximal hit rate. Consequently, proxy cache contents must be saved across system failures. At the same time, recovery must be quick. With today's disk sizes, the system cannot wait for full disk scans before servicing document requests. Hummingbird warms the main memory cache with files and clusters that were "hot" before the system reboot. Bounding file create and delete persistence rather than attaching them to system call semantics allows for higher performance.

Data stored on disk. Hummingbird's disk storage is segmented into four regions:

- clusters, which store the file data and meta-data,
- mappings of files to clusters, which allow a file to be quickly located on disk by identifying which clusters the file is in,
- the hot cluster log, which caches frequently used clusters, and
- the delete log, which stores small records describing intentional deletes.

The mappings of files to clusters is part of the file meta-data described in Section 3.2.

Warming the cache. Hummingbird lacks a directory structure and all meta-data consistency dependencies are contained within clusters. There is no need for an analog of the UNIX `fsck` utility to ensure file system consistency after a crash. During a planned shutdown, Hummingbird will write the file-to-cluster mappings to disk. (They are also written periodically.) During a crash, the system has no such luxury. So, while the file system is guaranteed to be consistent with itself, the lack of directories might make it impossible to locate files on disk immediately after a crash if the file-to-cluster mapping was not up to date. Hummingbird speeds up crash recovery time with a log containing the cluster identifiers of popular clusters. The `sync_hot_clusters_daemon()` creates this log using the cluster LRU list and writes it to disk periodically. After a crash, these clusters are scanned in first, quickly achieving a hit rate close to that before the crash.

Persistence. Hummingbird does not change disk contents immediately for file and cluster deletions. Research with journalling file systems has shown that hard meta-data update persistence is expensive, due to the necessity for updating stable storage synchronously [22]. Hummingbird does not provide hard persistence, but uses a log of intentional deletions to bound the persistence of deletions. Records describing deletions, either cluster or file, are written into the log, which is buffered in memory. Periodically, the log is written out to disk according to the specified thresholds. The user specifies when the log should be written by specifying either a number of files threshold (i.e., once X files have been recorded in the log, it must be written to disk), or a threshold on the passing of time (i.e., the log must be written to disk at least once every Y seconds).

The log is structured as a table, indexed by cluster or file identifier. When a file or cluster is overwritten on disk, the delete intent record is removed from the log. Records contain file or cluster identifiers as well as a generation number, which is stored in the on-disk meta-data and used to eliminate the possibility of replayed deletes.

The recovery procedure. During recovery, all four regions of the disk are locked exclusively by the recovery process. The Hummingbird interface comes alive to the application early in the process – after the hot cluster log of the previous session has been recovered. However, Hummingbird will not write new files or clusters to disk until recovery has completed; `write_file()` calls will fail. It

continues to recover the clusters in the background and rebuild the in-memory meta-data. During this phase, requests that do not hit in the currently-rebuilt meta-data are checked in the file-to-cluster mappings to identify the cluster that needs to be recovered. Since the file-to-clusters mapping can be out-of-date (due to a crash), a file without a file-to-cluster mapping is viewed as not in the file system.

We now outline the sequence of events during crash recovery; recovery is much simpler during a planned shutdown. (1) Crash or power failure. The system reboots and enters recovery mode. (2) The hot cluster log is scanned, the hot clusters are read in, and their contents are used to initialize in-memory meta-data. (3) The file-to-cluster mappings are read in from disk. (4) The delete log is scanned and records are applied, modifying meta-data as necessary. (5) Proxy service is enabled. Hummingbird will now service requests. (6) During idle time, the recovery process scans the cluster region, rebuilding the in-memory meta-data for all files and clusters. As files are recovered, they are available to the application. (7) Recovery is complete. All files and clusters are now available. Hummingbird can now write new clusters to disk.

3.5 Daemons

Hummingbird employs a number of daemons in addition to the recovery daemons mentioned above that run when the file system is idle and can be called on demand. In the future, we hope to support client-provided daemons, which would, for example, support different cache eviction policies.

Pack files daemon. If the amount of main memory used to store files exceeds a tunable threshold, the `pack_files_daemon()` uses the file LRU list to create a cluster of files and write the cluster to disk. The daemon packs the files using the information from the `collocate_file()` calls, attempting to pack files from the same locality set in the same cluster. If a file is larger than the cluster size, it is split between multiple clusters.

This daemon uses the *disk data layout policy* to decide which cluster to pack next. Implemented policies include: *closest cluster*, which picks the closest free cluster to the previous cluster on disk accessed, *closest cluster to previous write*, which picks the closest free cluster to the previous cluster written to on disk, and *overwrite*, which overwrites the next cluster. All of these policies tend to write files accessed within a short time to clusters close together on disk. Thus, long-term fragmentation does

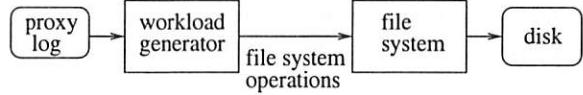


Figure 1: Simulation environment.

not occur.

Pack files from head daemon. The `pack_files_daemon()` takes files off the tail of the LRU list to pack into clusters; `pack_files_from_head_daemon()` takes files off the head. This has the effect of packing the most-frequently accessed files into clusters.

Free main memory data daemon. This daemon evicts data using file and cluster LRU lists when the amount of main memory used by the file system exceeds a tunable threshold.

Free disk space daemon. This daemon deletes files or clusters whose age exceeds tunable thresholds set by the *delete file policy* and the *delete cluster policy*.

4 Experimental results

We built an environment to run trace-driven experiments on real implementations of UFS and Hummingbird. This environment consists of four components, as depicted in Figure 1. (1) The processed *proxy log* which was discussed in Section 2.1. (2) The *workload generator* simulates the operation of a caching web proxy. It reads the proxy log events and generates the file system operations that a proxy would have generated during processing of the original HTTP requests. (3) The *file system*, which is either Hummingbird or various implementations of UFS, EFS, or XFS. (4) The *disk*, which is a physical magnetic disk that the file system uses to store the files. The workload generators, implementation details, and experiments are described in the following section.

4.1 Workload generators

We developed two workload generators: `wg-Squid` which mimics Squid's interaction with the file system, and `wg-Hummingbird` which issues Hummingbird calls. The same `wg-squid` workload generator was used with the various UFS implementation, EFS and XFS. The generators take as input the modified proxy access log. The workload generators operate in a trace-driven loop which processes

logged events sequentially without pause². This simulates a heavily loaded server.

UFS workload generator: **wg-Squid.** The **wg-Squid** simulates the file system behavior of Squid-2.2. The Squid cache has a 3-level directory hierarchy for storing files; the number of children at each level is a configurable parameter. In order to minimize file lookup times, Squid attempts to keep directories small by distributing files evenly across the directory hierarchy.

When a file is written to the cache a top-level directory, or *SwapDir*, is selected. Squid attempts to load balance across the SwapDirs. Once the SwapDir has been selected, the file is assigned a file number which uniquely identifies the file within the SwapDir. The value of this file number is used to compute the names of the level-2 and level-3 directories. Thus, Squid does not use the URL or URL reference locality for file placement into directories, limiting the ability of the file system to collocate files which will be accessed together. Once the directory path has been determined, the file is created and written.

Squid has configurable high and low water marks. When the total cache size passes the high water mark, eviction begins. Files are deleted from the cache in modified LRU order with a small bias towards expired files. Eviction continues until the low water mark is reached. The **wg-Squid** simulates this behavior except that it uses LRU instead of modified LRU; our log does not contain the expires information.

Hummingbird workload generator: **wg-Hummingbird.** Each iteration of the **wg-Hummingbird** loop parses the next sequential event from the log and attempts to read the URL from Hummingbird. If successful, it explicitly releases the file buffer. If the read attempt fails, it attempts to write the URL into the file system; this would have occurred after the proxy fetched the URL from the server.

The **wg-Hummingbird** maintains a hash table keyed by client IP address to store information for generating the collocate hints. The values in the hash table are the URLs of the most recent

²We call this timing mode THROTTLE. We have implemented two other different timing modes: REAL and FAST. REAL issues the event with the frequency that they were recorded. FAST processes the events with a speed-up heuristic parameterized by n ; if the time between events is longer than time n , then we only wait time n between them.

HTML file request seen from a particular client. The hash table only stores the URLs of static HTML files. As requests for non-HTML documents are processed, **wg-Hummingbird** generates `collocate_files()` calls for the non-HTML paired with its client's current HTML file, as stored in the hash table.

Note that Squid (and **wg-Squid**) do not provide explicit locality hints to the underlying operating system; they only place files in the directory hierarchy. This is in contrast with **wg-Hummingbird**, which provides explicit locality hints via the `colocate_files()` call. There are other ways which a proxy could use UFS to obtain file locality; we did not test against these other approaches. Thus, our comparisons are restricted to the Squid approach for file locality.

4.2 Experiments

We performed our experiments on PCs with a 700 MHz Pentium III running FreeBSD 4.1 with an 18 GB IBM 10,000 RPM Ultra2 LVD SCSI (UltraStar 18LZX). We also performed experiments on an SGI Origin 2000 with 18 GB 10,000 RPM Seagate Cheetah disks (ST118202FC) under IRIX 6.5. The results with different parameter settings were similar to each other so we only present a representative subset of the results.

Our experiments used the 4-day web proxy log as input into the workload generators. Among other measurements, we measured the *proxy hit rate*, which represents how frequently the web page will not have to be fetched from the server, and the *file system read time*, which represents how long the proxy must wait to get the file from the file system. The *file system write time* is the time it takes the file system to return after a write; this may not include the time to write the file or file metadata to disk. (We call the file system read (write) times *FS read (write) time* in our figures and tables.) **wg-Squid** and **Hummingbird** measured the file system read/write times directly, and the output of the `iostat -o` and `sar` commands were used to determine the disk I/O times. We also report the *throughput*, which we compute as the ratio of the experiment run time and the total number of requests processed. The measurements are averaged over the entire log; warm-up effects were insignificant.

Comparing Hummingbird with UFS: single thread. We compared **Hummingbird** with three versions of UFS on FreeBSD 4.1. The three versions of UFS were: UFS, which is UFS mounted

Table 2: Comparing Hummingbird with UFS, UFS-async, and UFS-soft when files greater than 64 KB are not cached.

file system	disk size	main memory (MB)	proxy hit rate	FS read time (ms)	FS write time (ms)	# of disk I/Os	mean disk I/O time (ms)
Hummingbird	4 GB	256	0.62	1.81	0.32	1,161,163	5.14
	4 GB	128+128	0.64	6.13	2.54	4,807,440	4.66
	4 GB	128+128	0.64	5.54	21.07	10,737,300	4.97
	4 GB	128+128	0.64	5.93	20.77	10,238,460	5.02
Hummingbird	4 GB	1024	0.63	1.05	0.03	552,882	5.75
	4 GB	512+512	0.64	3.21	2.35	3,217,440	3.95
	4 GB	512+512	0.64	3.27	20.85	9,714,420	4.76
	4 GB	512+512	0.64	3.92	18.43	9,022,200	4.63
Hummingbird	8 GB	256	0.64	2.03	0.32	1,194,494	5.64
	8 GB	128+128	0.67	5.69	1.47	4,605,360	4.34
	8 GB	128+128	0.67	5.78	15.66	9,058,141	4.86
	8 GB	128+128	0.67	6.17	12.83	8,313,360	4.63
Hummingbird	8 GB	1024	0.64	1.20	0.03	578,562	6.37
	8 GB	512+512	0.67	4.05	1.34	3,561,180	4.13
	8 GB	512+512	0.67	3.74	15.65	8,148,721	4.62
	8 GB	512+512	0.67	3.81	15.70	7,611,840	4.68

synchronously (the default), UFS-soft, which is UFS with soft updates, and UFS-async, which is UFS mounted asynchronously, so that meta-data updates are not synchronous and the file system is not guaranteed to be recoverable after a crash. We used a version of Hummingbird with a single working thread, where the daemons were called explicitly every 1000 log events. Table 2 presents comparisons for two different disk sizes, 4 GB and 8 GB, with two memory sizes, 256 MB and 1024 MB, when files greater than 64 KB are not cached. The memory was split evenly between the Squid cache and the file system buffer cache³. The proxy-perceived latency in Table 2 is the 5th column, the FS read time. Hummingbird’s smaller file system read time is due to the hits in main memory caused by grouping files in locality sets into clusters. Hummingbird’s smaller file system write time (6th column) when compared to UFS-async is due to cluster writes, which write multiple files to disk in a single operation. The FS write times for UFS and UFS-soft are greater than UFS-async due to the synchronous file create operation.

The effectiveness of the clustered reads and writes and the collocation strategy is illustrated in the number of disk I/Os. In all test configurations, Hummingbird issued substantially fewer disk I/Os than any of the UFS configurations. Also, note that

³We controlled the size of the file system buffer cache by locking the Squid’s memory using the `mlock` system call and locking additional memory, so that file system buffer cache could use only the remaining unlocked memory. In this way we also prevented paging of the Squid process.

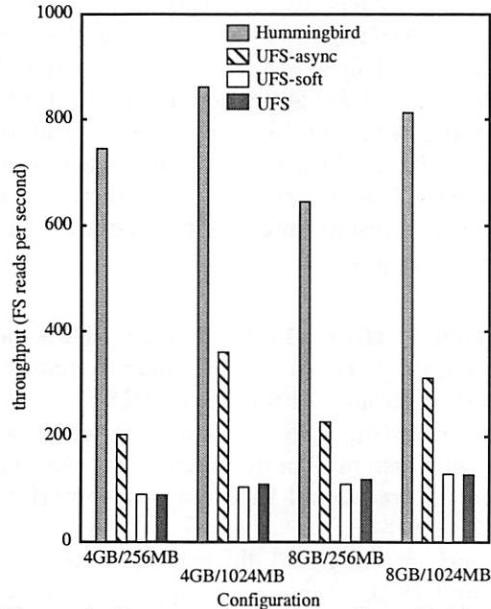


Figure 2: Request throughput from Table 2.

the number of disk I/Os in the UFS experiments is larger than the total number of requests in the log. This is because file operations resulted in multiple disk I/Os. This also explains why UFS read and write operations (as seen in FS read and write times) are slower than individual disk I/Os. The mean disk I/O time is larger in Hummingbird since the request size is a cluster, which is larger when compared to the mean data transfer size accessed by UFS.

The throughput for each experiment in Table 2 is

shown in Figure 2. Figure 2 shows that Hummingbird throughput is much higher than both UFS, UFS-soft, and UFS-async on the same disk size and memory size. This is not quite a fair comparison since the proxy hit rate is lower with Hummingbird. (We do not expect the experiment run time to increase more than 10% when the Hummingbird policies are set so that it would have equivalent hit rate to *wg-Squid*). The throughput is larger for Hummingbird since much less time is spent in disk I/O. Using throughput as a comparison metric, we see that Hummingbird is 2.3–4.0 times faster than simulated Squid running on UFS-async, 5.6–8.4 times larger than a simulated version of Squid running on UFS-soft, and 5.4–9.4 times faster than simulated Squid running on UFS. These numbers include also the results from Table 3.

The experiments for Table 2 assumed that files greater than 64 KB were not cached by the proxy. We got similar results when assuming the proxy would cache all files; see Table 3. Note that the proxy hit rate in Table 3 is lower than in Table 2. This is the result of the cache being “polluted” with large files, which cause some smaller files to be evicted. The end result is that there are less hits, which translate into less file system activity, and fewer file accesses.

Comparing Hummingbird with EFS and XFS: single thread. We compared Hummingbird with EFS and XFS on SGI IRIX. EFS is an extent file system. XFS is a high-performance journaling file system that interacts with the kernel through the traditional VFS and vnode interfaces.

We experimented with 3 different disk sizes, 4 GB, 9 GB, and 18 GB. Since EFS supports file systems up to 8 GB in size, we could not test it with 9 or 18 GB disks. Table 4 presents results where *wg-Squid* cache and the IRIX buffer cache together use 256 MB of main memory, which are divided in two ways: 50 MB + 206 MB and 128 MB + 128 MB (*wg-Squid* and buffer cache, respectively). The 50 MB for the *wg-Squid* was selected according to the Squid administration guidelines [23]. Hummingbird has 256 MB of main memory, and it has the best choice of policies as previously discussed.

Our experimental results are in Table 4. Hummingbird throughput is much higher than both XFS and EFS on the same disk size as seen in the UFS experiments, and the user-perceived latency (5th column in Table 4) is smallest in Hummingbird. The sec-

ond observation from Table 4 is that EFS is much slower than XFS, which is a journaling file system. In particular, file system write time for EFS is more than 3 times larger than XFS write time. It is the result of frequent synchronous write operations for meta-data, which are performed by EFS. The third observation is that increasing *wg-Squid* cache size to 128MB and reducing the file system buffer cache size actually improved the file system performance, which indicates that the *wg-Squid* cache is more effective than the file system buffer cache.

Comparing Hummingbird with UFS: multi-threaded workload generators. Many proxies are either multi-threaded or have multiple processes. Hummingbird is both thread- and process-safe. We implemented multi-threaded versions of both our workload generators, with a thread for the daemons in *wg-Hummingbird*, and ran similar experiments to the above with one processor running FreeBSD 4.1 with the LinuxThreads library. Table 5 contains a subset of the results of our experiments using four threads and two disks. In multi-threaded squid workload generator, two cache root directories are used, each residing on a disk. The experiment run time for the experiments in Table 5 are consistently longer than the corresponding cases in Table 3 due to an uneven distribution of the files on the 2 disks; we observed a bursty access pattern to each disk in the iostat log. Queuing in the device driver results in longer FS read/write time too. However, the Hummingbird throughput is again 2–4 times greater than for UFS, UFS-async, and UFS-soft.

Recovery performance. Recovering the hot clusters and the deletion log are quick; it takes about 30 seconds to read 2500 hot clusters and a log of 3000 deletion entries into memory. Thus, on a system crash with a 18 GB disk, Hummingbird can start to service requests in 30 seconds, while UFS will take more than 20 minutes to perform the necessary fsck before requests can be serviced. While rebuilding the in-memory meta-data (Step 6 in Section 3.4), the FS read time increased a small amount (e.g., from 2.03 ms to 2.50 ms for a 8 GB disk).

Journaling file systems will recover integrity quickly compared with traditional UFSs due to log-based recovery. However, journaling alone will not fetch hot data from disk as part of the recovery process.

5 Polygraph results

A web cache benchmarking package called Web Polygraph [18] is used for comparison of caching

Table 3: Comparing Hummingbird with UFS, UFS-async, and UFS-soft when all files are cached.

file system	disk size	main memory (MB)	proxy hit rate	FS read time (ms)	FS write time (ms)	# of disk I/Os	mean disk I/O time (ms)	experiment run time (s)
Hummingbird	4 GB	256	0.60	1.68	0.39	1,349,175	4.17	6,362
	4 GB	128+128	0.62	6.61	2.88	5,134,380	4.72	25,510
	4 GB	128+128	0.62	5.81	22.91	11,858,100	5.02	59,475
	4 GB	128+128	0.62	6.13	22.67	11,283,060	5.05	59,807
Hummingbird	4 GB	1024	0.61	0.88	0.52	630,362	5.62	6,030
	4 GB	512+512	0.62	3.67	2.70	3,919,620	3.98	16,384
	4 GB	512+512	0.62	3.51	22.89	10,868,700	4.84	52,377
	4 GB	512+512	0.62	4.24	20.23	10,115,400	4.69	49,749
Hummingbird	8 GB	256	0.64	2.17	0.40	1,464,727	5.03	7,923
	8 GB	128+128	0.66	6.42	2.30	5,017,920	4.67	24,657
	8 GB	128+128	0.66	6.14	19.31	10,461,841	4.98	51,797
	8 GB	128+128	0.66	7.37	16.42	9,954,540	4.89	51,062
Hummingbird	8 GB	1024	0.62	1.18	0.51	695,771	6.41	6,802
	8 GB	512+512	0.66	4.66	1.90	4,016,400	4.32	18,240
	8 GB	512+512	0.67	3.69	15.71	8,158,080	4.61	37,097
	8 GB	512+512	0.66	4.24	18.90	8,894,760	4.82	45,044

Table 4: Comparing Hummingbird with XFS and EFS with 256 MB of main memory when all files are cached.

file system	disk size	cache size (MB)	proxy hit rate	FS read time (ms)	FS write time (ms)	# of disk I/Os	mean disk I/O time (ms)	experiment run time (s)
Hummingbird	4 GB	256	0.62	2.82	0.20	922,871	9.85	9,926
	4 GB	50+206	0.62	14.79	46.38	10,784,969	10.99	128,878
	4 GB	50+206	0.62	14.97	16.56	10,870,353	6.67	75,565
	4 GB	128+128	0.62	14.08	48.33	10,540,778	11.37	130,359
	4 GB	128+128	0.62	14.90	13.95	10,115,369	6.72	70,746
Hummingbird	9 GB	256	0.66	3.30	0.21	1,028,922	10.77	11,861
	9 GB	50+206	0.66	15.47	12.39	9,558,954	7.02	69,929
	9 GB	128+128	0.66	15.65	8.80	8,737,878	7.12	64,726
Hummingbird	15 GB	256	0.67	3.71	0.21	1,049,121	11.92	13,313
	15 GB	50+206	0.67	16.71	5.73	8,103,576	7.60	63,371
	15 GB	128+128	0.67	15.91	5.79	7,841,184	7.55	60,943

Table 5: Comparing Hummingbird with UFS, UFS-async, and UFS-soft with 256 MB of main memory when all files are cached with 2 disks and 4 threads in the workload generator.

file system	disk size	proxy hit rate	FS read time (ms)	FS write time (ms)	# of disk I/Os	mean disk I/O time (ms)	experiment run time (s)
Hummingbird	4 GB	0.64	4.86	1.34	1,478,005	11.68	11,743
	4 GB	0.66	5.92	6.39	5,638,201	10.72	30,427
	4 GB	0.66	3.25	20.42	9,405,301	9.76	45,655
	4 GB	0.66	6.71	15.86	10,771,201	9.05	48,577
Hummingbird	8 GB	0.67	6.12	1.40	1,556,169	14.08	12,997
	8 GB	0.67	6.19	5.18	5,466,061	10.67	29,354
	8 GB	0.67	6.04	14.67	8,678,761	10.52	44,622
	8 GB	0.67	6.78	9.73	8,903,641	8.74	38,464

web proxies. The clients and servers are simulated; the client workload parameters such as hit ratio, cacheability, and response sizes can be specified and server-side delays can be specified. We used the PolyMix-2 traffic model to compare Apache [25] using UFS and a slightly modified version of Apache using Hummingbird without `collocate_files()` calls. Due to space considerations of this paper, we only briefly discuss our results.

We used one of our FreeBSD Pentium IIIs for the proxy. We used a number of different client request rates; the following results are for 8 requests/second. We found that the mean response time for hits in the proxy is 14 times smaller with Hummingbird than with UFS, and the median response time for hits is 20 times smaller. The improvement in mean and median response times for proxy misses was much smaller, as expected; Hummingbird is 20% faster than UFS. Since Hummingbird serves proxy hits faster, its request queue is shorter, which in turn, shortens the queue time of proxy misses.

6 Related work

Related work falls into two categories: first, analyses of traditional UFS-based systems and ways to beat their performance limitations, and second, analyses of the behavior of web proxies and how they can better use the underlying I/O and file systems.

The first set of research extends back to the original FFS work of McKusick et al. [13] which addressed the limitations of the System V file system by introducing larger block sizes, fragments, and cylinder groups. With increasing memory and buffer cache sizes, UNIX file systems were able to satisfy more reads out of memory. The FFS clustering work of McVoy and Kleiman [14] sought to improve write times by lazily writing the data to disk in contiguous extents called *clusters*. LFS [20] sought to improve write times by packing dirty file blocks together and writing to an on-disk log in large extents called *segments*. The LFS approach necessitates a cleaner daemon to coalesce live data and free on-disk segments. As well, new on-disk structures are required. Work in soft updates [6] and journalling [7, 4] has sought to alleviate the performance limitations due to synchronous meta-data operations, such as file create or delete, which must modify file system structures in a specified order. Soft updates maintains dependency information in kernel memory to order disk updates. Journalling systems write meta-data updates to an auxiliary log using

the write-ahead logging protocol. This differs from LFS, in which the log contains all data, including meta-data. LFS also addresses the meta-data update problem by ordering updates within segments.

The Bullet server [26, 24] is the file system for Amoeba, a distributed operating system. The Bullet service supports entire file operations to read, create, and delete files. All files are immutable. Each file is stored contiguously, both on disk and in memory. Even though the file system API is similar to Hummingbird, the Bullet service does not perform clustering of files together, so it would not have the same type of performance improvement that Hummingbird has for a caching web proxy workload.

Kaashoek et al. [9] approaches high performance through developing *server operating systems*, where a server operating system is a set of abstractions and runtime support for specialized, high performance server applications. Their implementation of the Cheetah web server is similar to Hummingbird in one way: collocating an HTML page and its images on disk and reading them from disk as a unit. Web servers' data storage is represented naturally by the UFS file hierarchy. This is not true for caching web proxies as discussed in Section 2.2.

CacheFlow [2] builds a cache operating system called CacheOS with an optimized object storage system which minimizes the number of disk seek and I/O operations per object. Unfortunately, details of the object storage are not public. The Network Appliance filer [8] is a prime example of combination of an operating system and a specialized file system (WAFL) inside a storage appliance. Novell [15] has developed the Cache Object Store (COS) which they state is 10 times more efficient than typical file systems; few details on the design are available. The COS prefetches the components for a page when the page is requested, leading us to believe that the components are not stored contiguously as they are in Hummingbird.

Rousskov and Soloviev [21] studied the performance of Squid and its use of the file system. Markatos et al. [12] presents methods for web proxies to work around costly file system file opens, closes, and deletes. One of their methods, LAZY-READS, gathers read requests n -at-a-time, and issues them all at the same time to the disk; results are presented when n is 10. This is similar to our clustering of locality sets, since a read for a cluster will, on average,

access 8 files. We feel that Hummingbird is a more general solution to the decreasing the effect of costly file system operations on a web proxy.

Maltzahn et al. [10] compared the disk I/O of Apache and Squid and concluded that they were remarkably similar. In a later paper [11], they simulated the operation of several techniques for modifying Squid, one of which was to use a memory-mapped interface to access small files. Other techniques improved the locality of related files based on domain names. This paper reported a reduction of up to 70% in the number of disk operations relative to unmodified Squid. An inherent problem using one memory-mapped file to access all small objects is that it cannot scale to handle a very large number of objects. Like Hummingbird, using memory-mapped files requires modification to the proxy code.

Pai et al. [17] developed a kernel I/O system called IO-lite to permit sharing of “buffer aggregates” between multiple applications and kernel subsystems. This system solves the multiple buffering problem, but, like Hummingbird, applications must use a different interface that supersedes the traditional UNIX read and writes.

7 Conclusions

This paper explores file system support for applications which can take advantage of the performance/persistence tradeoff. Such a file system is especially useful for local caching of data, where permanent storage of the data is available elsewhere. A caching web proxy is the prime example of an application that may benefit from this file system.

We implemented Hummingbird, a light-weight file system that is designed to support caching web proxies. Hummingbird has two distinguishing features: it stores its meta-data in memory, and it stores groups of related objects (e.g., HTML page and its embedded images) together on the disk. By setting the tunable parameters to achieve persistence, Hummingbird can also be used to improve the performance of web servers, which have similar reference locality as proxies. Our results are very promising; Hummingbird’s throughput is 2.3–4.0 times larger than a simulated version of Squid running UFS mounted asynchronously on FreeBSD, 5.4–9.4 times larger than a simulated version of Squid running UFS mounted synchronously on FreeBSD, 5.6–8.4 times larger than a simulated

version of Squid running UFS with soft updates on FreeBSD, and 5.4–13 times larger than XFS and EFS on IRIX. The Web Polygraph environment confirmed these improvements for the response times for proxy hits.

Additional information about Hummingbird is available at <http://www.bell-labs.com/project/hummingbird/>.

Acknowledgements. Many thanks to Arthur Goldberg for giving us copies of the ISP log files that we studied. WeeTeck Ng helped us with preparing our experiment environment. Bruce Hillyer and Philip Bohannon were very helpful in initial design discussions. Hao Sun performed the Polygraph experiments.

References

- [1] BAKER, M. G., HARTMAN, J. H., KUPFER, M. D., SHIRRIFF, K. W., AND OUSTERHOUT, J. K. Measurements of a distributed file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Asilomar (Pacific Grove), CA, Oct. 1991), ACM Press, pp. 198–212.
- [2] Network cache performance measurements. CacheFlow White Papers Version 2.1, CacheFlow, Sept. 1998.
- [3] CAO, P., AND IRANI, S. Cost-aware WWW proxy caching algorithms. 193–206.
- [4] CHUTANI, S., ANDERSON, O. T., KAZAR, M. L., LEVERETT, B. W., MASON, W. A., AND SIDEBOTHAM, R. N. The Episode file system. In *Proceedings of the Winter 1992 USENIX Conference* (San Francisco, CA, Winter 1992), pp. 43–60.
- [5] GABBER, E., AND SHRIVER, E. Let’s put NetApp and CacheFlow out of business! In *Proceedings of the 9th ACM SIGOPS European Workshop* (Kolding, Denmark, Sept. 2000), pp. 85–90.
- [6] GANGER, G. R., AND PATT, Y. N. Metadata update performance in file systems. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Monterey, CA, Nov. 1994), pp. 49–60.
- [7] HAGMANN, R. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (Austin, TX, Nov. 1987), pp. 155–162. In *ACM Operating Systems Review* 21:5.
- [8] HITZ, D., LAU, J., AND MALCOLM, M. File system design for an NFS file server appliance. In

- Proceedings of the USENIX 1994 Winter Technical Conference* (San Francisco, CA, Jan. 1994), pp. 235–246.
- [9] KAASHOEK, M. F., ENGLER, D. R., GANGER, G. R., AND WALLACH, D. A. Server operating systems. In *Proceedings of the 1996 SIGOPS European Workshop* (Connemara, Ireland, Sept. 1996), pp. 141–148.
- [10] MALTZAHN, C., RICHARDSON, K., AND GRUNWALD, D. Performance issues of enterprise level proxies. In *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems (Sigmetrics '97)* (Seattle, WA, June 1997), pp. 13–23.
- [11] MALTZAHN, C., RICHARDSON, K. J., AND GRUNWALD, D. Reducing the disk I/O of web proxy server caches. In *Proceedings of the 1999 USENIX Annual Technical Conference* (Monterey, CA, June 1999), pp. 225–238.
- [12] MARKATOS, E. P., KATEVENIS, M. G., PNEVMATIKOS, D., AND FLOURIS, M. Secondary storage management for web proxies. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems* (Boulder, CO, Oct. 1999), pp. 93–114.
- [13] MCKUSICK, M. K., JOY, W. N., LEFFLER, S. J., AND FABRY, R. S. A fast file system for UNIX. *ACM Transactions on Computer Systems* 2, 3 (Aug. 1984), 181–197.
- [14] MCVOY, L. W., AND KLEIMAN, S. R. Extent-like performance from a UNIX file system. In *Proceedings of the Winter 1991 USENIX Conference* (Dallas, TX, Jan. 1991), pp. 33–43.
- [15] The Novell ICS advantage: Competitive white paper. Tech. rep. Available at <http://www.novell.com/advantage/nics-compwp.html>.
- [16] OUSTERHOUT, J. K., DA COSTA, H., HARRISON, D., KUNZE, J. A., KUPFER, M., AND THOMPSON, J. G. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of 10th ACM Symposium on Operating Systems Principles* (Orcas Island, WA, Dec. 1985), vol. 19, ACM Press, pp. 15–24.
- [17] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. IO-lite: a unified I/O buffering and caching system. In *Proceedings of the Third USENIX Symposium on Operating System Design and Implementation (OSDI'99)* (New Orleans, LA, Feb. 1999), pp. 15–28.
- [18] POLYTEAM. Web polygraph site. <http://polygraph ircache.net/>.
- [19] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. A comparison of file system workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)* (San Diego, CA, June 2000), pp. 41–54.
- [20] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems* 10, 1 (Feb. 1992), 26–52.
- [21] ROUSSKOV, A., AND SOLOVIEV, V. A performance study of the Squid proxy on HTTP/1.0. *World-Wide Web Journal, Special Edition on WWW Characterization and Performance and Evaluation* 2, 1–2 (1999).
- [22] SELTZER, M. I., GANGER, G. R., MCKUSICK, M. K., SMITH, K. A., SOULES, C. A. N., AND STEIN, C. A. Journaling versus soft updates: asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference (USENIX-00)* (San Diego, CA, June 2000), pp. 71–84.
- [23] 1999. <http://squid.nlanr.net/mail-archive-squid-users/>.
- [24] TANENBAUM, A. S., VAN RENESSE, R., VAN STAVEREN, H., SHARP, G. J., MULLENDER, S. J., JANSEN, J., AND VAN ROSSUM, G. Experiences with the Amoeba distributed operating system. *Communications of the ACM* 33 (Dec. 1990), 46–63.
- [25] THE APACHE SOFTWARE FOUNDATION. Apache http server project. <http://www.apache.org/-httpd.html>.
- [26] VAN RENESSE, R., TANENBAUM, A. S., AND WILSCHUT, A. N. The design of a high-performance file server. In *Proceedings of the Ninth International Conference on Distributed Computing Systems (ICDCS 1989)* (Newport Beach, CA, June 1989), IEEE Computer Society Press, pp. 22–27.

Pragmatic nonblocking synchronization for real-time systems

Michael Hohmuth

Hermann Härtig

Dresden University of Technology

Department of Computer Science

drops@os.inf.tu-dresden.de, <http://os.inf.tu-dresden.de/drops/>

Abstract

We present a pragmatic methodology for designing non-blocking real-time systems. Our methodology uses a combination of lock-free and wait-free synchronization techniques and clearly states which technique should be applied in which situation.

This paper reports novel results in various respects: We restrict the usage of lock-free mechanisms to cases where the widely available atomic single-word compare-and-swap operation suffices. We show how Brinch Hansen's monitors (alias Java's synchronized methods) can be implemented on top of our mechanisms, thereby demonstrating their versatility. We describe in detail how we used the mechanisms for a full reimplementation of a popular microkernel interface (L4). Our kernel—in contrast to the original implementation—bounds execution time of all operations. We report on a previous implementation of our mechanisms in which we used Massalin's and Pu's single-server approach, and on the resulting performance, which lead us to abandon this well-known scheme.

Our microkernel implementation is in daily use with a user-level Linux server running a large variety of applications. Hence, our system can be considered as more than just an academic prototype. Still, and despite its implementation in C++, it compares favorably with the original, highly optimized, non-real-time, assembly-language implementation.

1 Introduction

In recent years, nonblocking data structures have caught the attention not only of the real-time systems community but of theoretical and some practical operating-

systems groups. Many researchers have devised new methods for efficiently synchronizing interesting data structures in a nonblocking fashion. Others have conceived general methodologies for transforming any algorithm into a nonblocking one; however, these results have a more theoretical nature as the methodologies often lead to very inefficient implementations. The next section briefly discusses a number of these works.

In contrast to this boom, we know of only a few system implementations that successfully exploit nonblocking synchronization. The only two operating systems we are aware of that use exclusively nonblocking synchronization are SYNTHESIS [16] and the CACHE kernel [7].

One of the problems with the approach is that it appears difficult to apply to many modern CPU architectures: Many of the most efficient algorithms available for lock-free data structures require a primitive for atomically updating two independent memory words (two-word compare-and-swap, CAS2), and many processors like the popular x86 CPUs do not provide such an instruction. Significantly, SYNTHESIS and the CACHE kernel originate from the Motorola 68K architecture, which does have a CAS2 primitive.

In this paper, we present a pragmatic approach for building nonblocking real-time systems. Our methodology works well even on CAS2-less architectures. It does not rely solely on lock-free synchronization for implementing nonblocking data structures—which would be both inconvenient and slow on the architectures we considered. Instead, our methodology does allow for locks, but ensures that the system is wait-free nonetheless. In addition, our technique is easy to apply because from a developer's perspective, it looks much like programming with mutual exclusion using monitors.

We describe the application of our approach to build a real system: Using our methodology, we developed the Fiasco microkernel, a kernel for the DROPS real-time operating system [8] that runs on x86 CPUs. This

kernel is an implementation of the L4 microkernel interface [15], and it is sufficiently mature to support all the software developed for L4, including DROPS servers and L⁴Linux [9].¹ We evaluate the effectiveness of our methodology for nonblocking design by examining the Fiasco microkernel’s real-time properties and synchronization overheads.

Fiasco currently runs only on uniprocessors. Consequently, we concentrate on single-processor implementation details. However, our methodology lends itself to multiprocessor-system implementations as well, and we point out routes for multiprocessor extensions.

We also discuss a number of nonblocking synchronization mechanisms. In their SYNTHESIS work, Masa-
lin and Pu [16] introduced the concept of a “single-server” thread (a variant of the “serializer” pattern first described by Hauser and associates [10]), which serializes complex object updates that cannot be implemented in a nonblocking fashion. In this paper, we present a simple modification to the single-server scheme that makes it truly nonblocking and useful for use in real-time systems. Furthermore, we show that the single-server mechanism is semantically equivalent to a locking scheme. In particular, the real-time version can be replaced by a locking scheme with priority inheritance that is easier to implement and has better performance.

We see our contribution as leading the recent interest in nonblocking synchronization to a practicable interim result, which the scientific community can verify. The source code to the Fiasco microkernel is freely available, allowing researchers to further study our techniques and experiment with them.

This paper is organized as follows: In Section 2, we consider related work on nonblocking synchronization. In Section 3, we develop our methodology for designing wait-free real-time systems. Section 4 shows how we applied this methodology to the development of the Fiasco microkernel. In Section 5, we present performance values for the Fiasco microkernel, and we evaluate the kernel’s real-time properties. In Section 6, we derive conditions for the applicability of our methodology for the development of multithreaded user-mode real-time programs. We conclude the paper in Section 7 with a summary and suggestions for future work.

¹L⁴Linux is a port of the Linux kernel (version 2.2.x) that runs as a user program on top of L4 and is binary compatible with original Linux.

2 Nonblocking synchronization and related work

2.1 Lock-free and wait-free synchronization

Overview. Nonblocking synchronization strategies have two important properties: First, they provide full preemptability and allow for multi-CPU concurrency. Second, priority inversion is avoided; lower-priority threads cannot block higher-priority threads because there is no blocking at all. These characteristics make nonblocking synchronization very interesting for real-time systems.

The concepts discussed in this section are not new in any way, and many systems implement variants of them such as optimistic concurrency control [1] and priority inheritance [20]. We describe them here for completeness.

Nonblocking synchronization comes in two flavors: wait-free and lock-free synchronization.

Wait-free synchronization can be thought of as locking, with helping replacing blocking. When a higher-priority thread *A*’s critical section detects an interference with a lower-priority thread *B*, *A* helps *B* to finish its critical section first. During helping, *A* lends *B* its priority to ensure that no other, lower-prioritized activities can interfere. When *B* has finished, *A* executes its own critical section.

Wait-free object implementations satisfy a stronger form of block-freedom than lock-free synchronization (discussed in the next paragraph) as they guarantee freedom from starvation. Therefore, many authors point out that wait-free synchronization is a special case of lock-free synchronization. However, wait-free synchronization can also be implemented using locks, albeit with a nonblocking helping scheme. For example, a locking scheme with priority inheritance can be considered a wait-free synchronization scheme as long as critical sections never block.

Lock-free synchronization works completely without locks. Critical code sections are designed such that they prepare their results out of line and then try to commit them to the pool of shared data using an atomic memory update instruction like compare-and-swap (CAS). The *compare* part of CAS is used to detect conflicts between two threads that simultaneously try to update the data; if it fails, the whole operation is restarted. If needed, retries

can be delayed with an exponential backoff to avoid retry contention.²

This synchronization mechanism has some nice properties: Because there are no locks, it avoids deadlocks; it provides better insulation from crashed threads, resulting in higher robustness and fault tolerance, because operations do not hold locks on critical data; moreover, it is automatically multiprocessing-safe.

Preconditions for using lock-free synchronization are that primitives for atomic memory modifications are available, and data is stored in type-stable memory management. We do not digress into type-stable memory management in this paper (see [7] for a discussion of operating-systems-related issues); the rest of this subsection discusses atomic memory modification.

Atomic memory update. The x86 CPUs have two kinds of atomic memory-modification operations: a test-and-set instruction (TAS) and a CAS instruction. Newer models (Intel Pentium and newer) also have a double-size-word (8 bytes) compare-and-swap instruction (CASW). However, these CPUs do not support atomically updating two independent memory words (two-word compare-and-swap, CAS2).

A number of data structures can be implemented without locks directly on top of CAS and CASW (i.e., without the overhead of a software-implemented multi-word CAS): counters and bitfields with widths up to 8 bytes, stacks, and FIFO queues. [21, 18]

Valois introduced a lock-free single-linked list design supporting insertions and deletions anywhere in a list, as well as several other data structures [23, 22]. These designs also work with just CAS. However, Greenwald [6] has criticized them for being quite complex, difficult to get right, and computationally expensive.

Most of the algorithms for lock-free data-structure synchronization that have been developed recently assume availability of a stronger atomic primitive like CAS2. These data structures include general single-linked and double-linked lists. [6]

A number of techniques exist for implementing lock-free and wait-free general multi-word compare-and-swap (MWCAS) on top of CAS and CAS2, enabling nonblocking synchronization for arbitrarily complex data structures [11, 19, 2, 6]. These techniques have con-

siderable overhead in both space and runtime complexity, especially when compared to common lock-based operations, making them less interesting for kernel design.

The most common technique to implement atomic multi-word updates on uniprocessors is to prevent preemption during the update. This is usually done by disabling interrupt delivery in the CPU. The disadvantage of this method is (of course) that it does not work on multiprocessors.

Bershad [4] has proposed to implement CAS in software using an implementation and lock known to the operating system. When preempting a thread, the operating system consults the lock, and if it is set, it rolls back the thread and releases the lock. Greenwald and Cheriton [7] discuss a generalization of this technique to implement CAS2 or MWCAS. This method has the disadvantage of incurring overhead for maintaining the lock. Also, on multiprocessors, the lock must be set even when reading from shared data structures because otherwise readers can see intermediate states.

Another technique to facilitate complex object updates is the “serializer” or “single-server” approach [10]. It uses a single server thread to serialize operations. Other threads enqueue messages into the server thread’s work queue to request execution of operations on their behalf. If the server thread runs at a high priority, it does not block the requesting thread any more than if it had executed the operation directly.

2.2 Nonblocking synchronization in operating systems

We know of two other operating system projects that have explored nonblocking synchronization in the kernel: the CACHE kernel [7] and SYNTHESIS [16].

Both systems run on architectures with a CAS2 primitive (the Motorola 68K CPU), and their authors found CAS2 to be sufficient to synchronize accesses to all of their kernel data structures. The authors report that lock-free implementation is a viable alternative for operating-system kernels.

Massalin and Pu [16] originally also implemented a single-server mechanism for use in their lock-free SYNTHESIS kernel, but later they found no need to use it; the same was true for Greenwald and Cheriton [7] in

²Backoff is never needed on single-CPU systems.

their CACHE kernel. We will revisit the single-server approach in Section 4.4.

Greenwald and Cheriton [7] report that they found a powerful synergy between nonblocking synchronization and good structuring techniques for operating systems. They assert that nonblocking synchronization can reduce the complexity and improves the performance, reliability, and modularity of software especially when there is a lot of communication in the system.

However, they also warn that their results may not be applicable if the CPU does not support a CAS2 primitive. In this paper, we will investigate how nonblocking systems can be implemented in such an environment.

2.3 Nonblocking synchronization vs. real-time systems

Nonblocking object implementations are of interest for real-time systems because they provide preemptability and avoid priority inversion. However, while it is well-known that wait-free method implementations are bounded in time (there is only a fixed number of threads we have to help; no retry loop), it is not immediately apparent that this also applies to lock-free synchronization. On the surface, lock-free methods (like the ones in Figure 3 in Section 4.3) look dangerous because of their potentially unlimited number of retries.

Fortunately, Anderson and colleagues [3] recently determined upper bounds for the number of retries that occur in priority-based systems. They derived scheduling conditions for hard-real-time, periodic tasks that share lock-free objects, and reported that lock-free shared objects often incur less overhead than object implementations based on wait-free or lock-based synchronization schemes.

3 A design methodology for real-time systems

3.1 Design goals

Our main design goal was to allow for good real-time properties of our systems. More specifically, we wanted higher-priority threads to be able to preempt the system (including the kernel) at virtually any time, as soon as

they are ready to run—thus allowing for good schedulability of event handlers [12]. This should be true for sets of threads that depend on common resources, but even more so for independent thread sets.

Secondary goals to the first one are that short critical sections working on global state should induce essentially no overhead for synchronization; also, the synchronization scheme should work for both single-processor and multi-processor architectures.³

Finally, the design should be applicable to x86-compatible uniprocessors, that is, it must be implementable without CAS2.

3.2 Design guidelines

The first design goal rules out any synchronization scheme that suffers from priority inversion. Therefore, we have been looking into nonblocking synchronization schemes: lock-free and wait-free synchronization.

The secondary goals strongly favor lock-free synchronization schemes: Locks induce overhead, and in the multi-CPU case, the CPUs would compete for the locks. We therefore generally disallow lock-based schemes for frequently-used global state except where we have no other way out.

In particular, our design methodology comprises the following guidelines:

We classify a system's objects as follows: *Local state* consists of objects used only by related threads, that is, threads that cooperate on a given job or assignment. *Global state* consists of the objects shared by unrelated threads.

Frequently-accessed global state should be implemented using data structures that can easily be accessed with lock-free synchronization.

In Section 2.1, we mentioned a number of data structures that can be synchronized in this fashion on x86 CPUs using only CAS: Counters, bitfields, stacks, and FIFO queues.

With the x86 CPU lacking anything better than single-word CAS, we suggest that other global data (like double-linked lists) are also implemented in a lock-free

³We will point out incompatibilities of our design methodology with multiprocessor architectures where they occur.

fashion, based on a software implementation of MW-CAS.

In a kernel, the atomic update can be protected by disabling interrupts as discussed in Section 2.1. Of course, disabling interrupts does not help on multiprocessors; there, we suggest using spin locks to protect very short critical sections.

We discuss software-MWCAS for user-mode programs in Section 6.

Global state not relevant for real-time computing, and local data can be accessed using wait-free synchronization. We propose a wait-free priority-inheritance locking mechanism that can be characterized as “locking with helping,” explained in more detail in Section 3.3. This kind of synchronization has some overhead. Therefore, it should be avoided for objects that otherwise independent threads must access.

In our synchronization scheme, waiting for events inside critical sections is not allowed. This restriction ensures wait-freedom. We will show in Section 3.3 that this restriction does not limit the synchronization mechanism’s power.

Once a designer has decided which object should be synchronized with which scheme, our methodology becomes very straightforward to use. It approximates the ease of use of programming with mutual exclusion using monitors while still providing the desired real-time properties.

3.3 Wait-free locking with helping

We suggest a wait-free locking-with-helping scheme. Each object to be synchronized in this fashion is protected by a lock with a “wait” stack, or more correctly, with a helper stack.

A lock knows which thread holds it upon entering its critical section. When a thread A wants to acquire a lock that is in use by a different thread B , it puts itself on top of the lock’s helper stack. Then, instead of blocking and waiting for B to finish, it helps B by passing the CPU to B , thereby effectively lending its priority to B and pushing B out of its critical section. Every time A is reactivated (because the previous time slice has been consumed, or because of some other reason), it checks whether it now owns the lock; if it does not, it continues to help B until it does. When B finishes its critical

section, it will find a helping thread on top of the lock’s stack—in this case, thread A —and passes the lock (and the CPU) to that thread.

Using a stack instead of a FIFO wait queue has an important advantage: Given that threads are scheduled according to hard priorities, it follows that the thread with the highest priority lands on top of the helper stack. There is no way for a lower-priority thread to get in front of a higher-priority thread: As the high-priority thread does not go to sleep after enqueueing in the helper stack, it cannot be preempted by a lower-priority thread and remains on top of the stack.⁴ This property ensures that the highest-priority threads get their critical sections through first. It makes our locking mechanism an implementation of priority inheritance.

Of course, execution of critical sections may be preempted by higher-priority threads that become ready to run in the meantime. However, to ensure wait-freedom, threads executing a critical section must not sleep or wait.

Instead, threads first must leave critical sections they have entered before they go to sleep. This requirement raises the question of how to deal with producer-consumer-like situations without race conditions. There are a number of textbook solutions for this problem. We describe our solution in Section 4.3.

As long as critical sections do not nest, it is easy to see that our construction can be used to implement wait-and-notify monitors⁵ [14] (or their recent descendant, Java synchronized methods). Whenever a monitor-protected object’s method is called, we acquire the object’s lock. The wait operation would then be implemented as an unlock–sleep–lock sequence. Figure 2 shows a possible monitor implementation that uses a simple lock-free semaphore, shown in Figure 1.

Synchronization is more difficult when more than one object can be locked at a time. We will discuss two scenarios: nested monitor calls (i.e., nested critical sections), and atomic acquisition of multiple locks.

⁴This is generally true only for uniprocessors. For multiprocessors, the priority ordering of the helper list could be ensured by using a different data structure—a priority queue—or by first migrating the helper to the CPU of the lock owner to force it into that CPU’s priority-based execution order. There are subtle arguments for both designs, which are beyond the scope of this work.

⁵There is a large variety of monitors with differing semantics, but most of them can be shown to have equivalent expressive power [13, 5]. Wait-and-notify monitors, also classified as “no-priority non-blocking monitors” [5], have first been used in Mesa [14].

```

class Binary_semaphore
{
    Thread_list d_q; // Lock-free thread list
    int         d_count;

public:
    void down ()
    {
        d_q.enqueue (current());

        int old;
        do
        {
            old = d_count;
        }
        while (! CAS (&d_count, old, old - 1));

        if (old > 0)
        {
            // Own the semaphore,
            // can safely dequeue myself
            d_q.dequeue (current());
        } else {
            sleep (Thread_sem_wakeup);
            // Have been dequeued in up ()
        }
    }

    void up ()
    {
        int old;
        do
        {
            old = d_count;
        }
        while (! CAS (&d_count, old, old + 1));

        if (old < 0)
        {
            Thread* t = d_q.dequeue_first ();
            wakeup (t, Thread_sem_wakeup);
        }
    }
}; // Binary_semaphore

```

Figure 1 Pseudocode for a simple lock-free binary semaphore (for single-CPU machines). It makes use of a lock-free list of threads (`Thread_list`) with a given queuing discipline, for example a FIFO queue or a priority queue, and `sleep` and `wakeup` primitives like those in Figure 3.

```

class Monitor
{
    Helping_lock d_lock;

public:
    void enter ()
    {
        d_lock.lock (); // Locking w/ helping
    }

    void leave ()
    {
        d_lock.unlock ();
    }

    void wait (Binary_semaphore* condition)
    {
        d_lock.unlock ();
        condition->down ();
        d_lock.lock (); // Locking w/ helping
    }

    void signal (Binary_semaphore* condition)
    {
        condition->up ();
    }
}; // Monitor

```

Figure 2 Pseudocode for a wait-and-notify monitor based on a helping lock. This is a simple textbook implementation—except that it uses only nonblocking synchronization primitives. Semaphores used as condition variables need to be initialized with 0.

The `signal` operation wakes up a waiter according to the semaphore's queueing discipline. When one or more waiters have been restarted, and more threads are trying to enter the monitor, the `Helping_lock`'s helper stack guarantees that the thread with the highest priority can enter the monitor first.

As long as monitor methods never wait for events, locking with helping works for nested monitor calls in the same way as for non-nested monitors. However, if a nested method wants to wait for an event, freeing the nested monitor does not help because the outer monitor would still be locked during the sleep—which is illegal under our scheme. That is why nested monitor calls must not sleep.

There are two ways to deal with this restriction: Either construct the system such that second-level monitors or even all monitors never sleep, or make the locking more coarse-grained so that all objects that would have to be locked before going to sleep are in fact protected by a single monitor.

In the Fiasco microkernel, we have chosen the first option; in fact, we constructed the kernel so that critical sections never need to sleep. We discuss synchronization in the Fiasco microkernel in more detail in Section 4.3.

A different situation arises if the locks a critical section needs are known before the critical section starts, and during its execution. In this case, the wait operation can release all locks before sleeping, and reacquire them afterwards.

4 Synchronization in the Fiasco microkernel

We developed the Fiasco microkernel as the basis of the DROPS operating-system project—a research project exploring various aspects of hard and soft real-time systems and multimedia applications on standard PC hardware [8]. The microkernel runs on uniprocessor x86 PCs, and it is an implementation of the L4/x86 binary interface [15]. It is able to run L⁴Linux [9], a Linux server running as a user-level program that is binary compatible with standard Linux, and it is freely available from <http://os.inf.tu-dresden.de/drops/>.

The kernel closely follows the design outlined in Section 3. In this section, we report how various data structures are synchronized in this kernel, and we detail the design of our wait-free locking-with-helping mechanism.

4.1 Kernel objects

Let us begin by briefly describing the objects the Fiasco microkernel implements. (For a philosophical discussion on what a microkernel should and should not implement, we refer to Liedtke [15].)

Local state

Threads. The thread descriptors contain the complete context for thread execution: a kernel stack, areas for saving CPU registers, a reference to an address space, thread attributes, IPC state, and infrastructure for locking (more on the latter in Section 4.3).

Address spaces. There exists one address space per task. Address spaces implement the x86 CPU’s two-level page tables. They also contain the task number, and the number of the task that has the right to delete this address space.

Hardware-interrupt descriptors. Each hardware interrupt can be attached to a user-level handler thread. The kernel sends this thread a message every time the interrupt occurs.

Mapping trees. Like L4, the Fiasco microkernel allows transferring persistent virtual-to-physical page mappings via IPC between tasks. The mapping in the receiving task is dependent on the sender such that when the mapping is flushed in the sender’s address space, mappings depending on it are recursively flushed as well [15]. Mapping trees are objects to keep track of these dependencies. There is one mapping tree per physical page frame.

Global state

Present list and ready list. These double-linked ring lists contain all threads that are currently known to the system, or ready-to-run, respectively. On both lists, the “idle” thread serves as start and end of the list.

Array of address space references. This array is indexed by an address space number. It contains a reference for each existing address space; for nonexistent address spaces, the array contains an address space index referring to the task that has a right to create the address space. The Fiasco microkernel uses this array for create-rights management, and to keep track of and look up created tasks.

Array of interrupt-descriptor references. In this array, the Fiasco microkernel stores assignments between user-level handler threads and hardware interrupts.

Page allocator. This allocator manages the kernel’s private pool of page frames.

Mapping-tree allocator. This allocator manages mapping trees. Whenever a mapping is flushed or transferred using IPC, the corresponding mapping tree grows or shrinks. Once certain thresholds are exceeded, a new (larger or smaller) mapping tree needs to be allocated; this behavior is an artifact of the Fiasco microkernel’s implementation of mapping trees.

4.2 Synchronization of kernel objects

Following our design methodology from Section 3.2, the global state should be synchronized using lock-free synchronization while for local state the overhead of wait-free locking is acceptable. Primarily, we closely adhered to these guidelines. But we also made the requirements somewhat stronger where performance is critical, and we allowed a small relaxation where it did not affect real-time properties.

Local state. Threads are the most interesting objects that must be synchronized. We accomplish synchronization using wait-free locks (described in Section 4.3). However, for IPC-performance reasons we do not lock all of a thread’s state. Instead, we defined some parts of thread data to be not under the protection of the lock, and use lock-free synchronization for accessing these parts. In particular, the following data members of thread descriptors are implemented lock-free: the thread’s state word, which also contains the ready-to-run flag and all condition flags for waiting for events (as explained in Section 3.3); and the sender queue, a double-linked list of other threads that want to send the thread a message. The state word can be synchronized using CAS. For the double-linked sender list we use a simulated MWCAS that disables interrupts during memory modification.⁶

The Fiasco microkernel protects mapping trees, like the bulk of the thread data, using wait-free locks.

Address spaces require very little synchronization. The kernel has to synchronize only when it enters a refer-

⁶For the prospective port of the kernel to SMP machines, we plan to protect this MWCAS using a spin lock per receiver.

ence to a new second-level page table into the first-level page table. Deletion does not have to be synchronized because only one thread can carry out this operation: Thread 0 of the corresponding task deletes it when it is itself deleted. Otherwise, we do not synchronize accesses to address spaces: Only a task’s threads can access the task’s address space, and the result of concurrent updates of a mapping at a virtual address is not defined. As mappings are managed in (concurrency-protected) mapping trees and not in the page tables, mappings cannot get lost, and all possible states after such a concurrent update are consistent.

We did not have to synchronize hardware-interrupt descriptors at all because once they have been assigned using their reference array (global state), only one thread ever accesses them.

Global state. The reference arrays for address spaces and hardware-interrupt descriptors can easily be synchronized using simple CAS.

For the double-linked present and ready lists, we had to resort to simulate MWCAS by disabling interrupts for a short time. These lists and the sender list mentioned previously were the only objects for which we had to revert to this “ugly” but inevitable synchronization method.⁷

We believe that it is unnecessary to implement the kernel allocators for pages and mapping trees with lock-free synchronization; here we used wait-free locking, as for the local state. We allowed this relaxation of our guidelines in these instances for the following reason: Threads with real-time requirements never allocate memory (for page tables) or shrink or grow mapping trees once they have initialized. Instead, they make sure that they allocate all memory resources they might need at initialization time. Therefore, real-time threads do not compete for access to these shared resources, and the overhead for accessing them is irrelevant. Should our assertion become untrue in the future, we will revisit this design decision.

4.3 Wait-free locking in the Fiasco microkernel

The implementation of wait-free locking with helping in the Fiasco microkernel is very similar to the mechanism presented in Section 3.3.

⁷For the SMP port, this does not present a problem: The ready list is per-CPU, so interrupt-disabling can still be used. Accesses to the present list are seldom and can be synchronized using a spin lock.

The Fiasco microkernel extends the basic wait-free locking mechanism in two respects.

First, thread locks in the Fiasco microkernel are furnished with a switch hint. This hint overrides the system's standard policy of scheduling the threads, locking thread or locked thread, once the locker frees the lock. Usually, the runnable thread with the highest priority wins, but the Fiasco microkernel's IPC system call semantics dictate that the receiver gets the CPU first. The hint is a flag that can take one of three values: When the lock is freed, switch to (1) the previously-locked thread, (2) the locker, or (3) to whoever has the higher priority. To achieve IPC semantics, the sender locks the receiver, wakes it up, and sets the hint to Value 1 before releasing the lock.

Second, when locking other objects (including threads), threads need to maintain a count of objects they have locked. This count is checked in the thread-delete operation to avoid deleting threads that still hold locks.

If one thread is locked by another, it usually cannot be scheduled. If the scheduler or some other thread activates a locked thread, its locker is activated instead. The only exception is an explicit context switch from a thread's locker. The thread-delete operation uses this characteristic to push to-be-deleted threads out of their critical sections.

The time-slice donation scheme introduced in Section 3.3 requires that nested critical sections do not sleep. During the implementation of the Fiasco microkernel, we did not find this limitation to be very restricting. We completely avoided nesting critical sections that might want to sleep: We found that even for complex IPC operations, there was no need to lock both of two interacting threads.

Instead, a thread *A* that needs to manipulate another thread *B* usually locks *B*, but not itself (*A*). Kernel code running in *A*'s context needs to ensure that locked operations on *A* itself (by a third thread, *C*) cannot change state that is needed during *A*'s locked operation on *B*. In practice, this is very easy to achieve: All locked operations first check whether a change to the locked thread is allowed. If the locked thread is not in the correct state, the locked operation is aborted. All threads explicitly allow a set of locked operations on them by adjusting their state accordingly.

Figure 3 shows pseudocode for our sleep and wakeup operations. As a means to avoid race conditions between sleep and wakeup, we use binary condition flags for syn-

```
void sleep (unsigned condition)
{
    Thread* thread = current ();

    for (;;)
    {
        unsigned old_state = thread->state;
        if (old_state & condition)
        {
            /* condition occurred */
            break;
        }
        if (CAS (& thread->state,
                 old_state,
                 old_state & ~Thread_running))
        {
            /* ready flag deleted, sleep */
            schedule ();
        }
        /* try again */
    }

    thread->state &= ~condition;
}

void wakeup (Thread* thread,
             unsigned condition)
{
    for (;;)
    {
        unsigned old_state = thread->state;
        if (CAS (& thread->state,
                 old_state,
                 old_state | Thread_running
                 | condition))
        {
            /* CAS succeeded */
            break;
        }
    }

    if (thread->prio > current()->prio)
        switch_to (thread);
}
```

Figure 3 Pseudocode for the sleep and wakeup operations. As the condition flag is stored in the same memory word as the scheduler's ready-to-run flag, the sleep implementation does not risk a race condition with the wakeup code.

chronization. All condition flags are located in the same memory word that also contains the scheduler’s ready-to-run flag. Using CAS, a thread that wants to sleep can make sure that the condition flag is still unset when it removes the ready-to-run flag.

This solution is only applicable inside a kernel, and it restricts the number of condition flags to the number of bits per memory word. For our microkernel, this was not a severe restriction (the Fiasco microkernel needs less than 10 condition flags), but it may become a problem for more complex systems. For such systems, a more general solution (e.g., protecting sleep and wakeup using a simple lock) can be used.

4.4 Single-server synchronization revisited

Before we implemented the wait-free locking scheme described in Section 4.3, we experimented with Massalin’s and Pu’s single-server synchronization scheme discussed in Section 2.1. In this section, we discuss how the single-server mechanism can be changed for real-time systems, and why we changed it into the simpler locking-with-helping scheme.

In Massalin’s and Pu’s scheme, threads that want to change an object put a change-request message into the request queue of the server thread that owns the object. In similar spirit to our helping-lock design from Section 3.3, we can minimize the worst-case wait time for high-priority threads by replacing the request queue with a stack (so that messages from high-priority senders get processed first), and by letting requesters actively donate CPU time to the server thread until it has handled their request.

When we first designed and implemented our wait-free synchronization mechanism, we drew inspiration from Massalin’s and Pu’s work. In particular, our design looked as follows:

Our kernel ensured serialization of critical sections by allowing only one thread, an object’s *owner*, to execute operations on that object. In other words, all locked operations ran in the thread context of the owner of an object.

Threads were their own owners. Consequently, threads carried out themselves all locked operations on them, including those initiated by other threads.

The kernel assigned ownership for other objects (not threads) on the fly using lock-free synchronization. This design can also be viewed as follows: The only object type that can be locked at all is the thread. All other objects are “locked” by locking a thread and assigning ownership of the object to that thread. Then, all operations on that object are carried out by the owner.

Helping an owner was as simple as repeatedly switching to the owner until either the owner had completed the request, or a thread that deleted the owner had aborted the request. The context-switching code took care of executing all requests before returning to the context of the thread.

We consider this design to be not inelegant, but unfortunately, it required a context switch for every locked operation. Only later we realized that this mechanism in fact shares many properties with the wait-free locking scheme with priority inheritance we derived in Section 3.3. Our new locking mechanism is less complex and performs much better than our original single-server scheme.

5 Performance evaluation

To evaluate the real-time properties of the Fiasco microkernel and the overhead of its synchronization mechanisms, we conducted two series of measurements. First, to verify that the kernel matches our requirements with regards to preemptability and scheduling, we measured the lateness of a user-level interrupt handler. Second, we measured the overhead of our synchronization primitives in a number of microbenchmarks.

5.1 Real-time characteristics

For this test, we set up a timer device to trigger a hardware interrupt every 250 µs. We created a user-level task containing a high-priority handler thread connected to the interrupt, and we measured the time between interrupt occurrences. From the results, we computed the maximum lateness. During measurements, a cache-flooding application and a Linux system running various multi-user benchmarks ran concurrently with the handler thread, inducing a high load on the system.⁸

⁸These measurements are equivalent to those Mehnert [17] carried out in 1999. Mehnert’s results showed a much worse maximum late-

System	Max. lateness
Fiasco µ-kernel / L ⁴ Linux	65 µs
L4/x86 / L ⁴ Linux	541 µs
RTLinux	58 µs

Table 1 Maximum lateness of a periodic 250-µs interrupt handler. On the Fiasco µ-kernel and on L4/x86, the handler ran in a user task of its own; in RTLinux, the handler was a real-time task running in kernel mode.

We carried out these measurements on a 200 MHz Pentium Pro machine. The CPU's built-in local APIC served as the interrupt source.

We conducted this test on three operating systems: on the Fiasco microkernel with L⁴Linux, on Liedtke's high-performance L4/x86 kernel with L⁴Linux, and, for comparison, on RTLinux [24] (with the handler running in kernel mode). Table 1 shows the maximum latenesses for the three systems. (The average lateness was very small—smaller than 1 µs on all systems.)

It turns out that maximum lateness in the Fiasco microkernel is an order of magnitude smaller than that for L4/x86. That is because L4/x86 uses interrupt disabling liberally throughout the kernel to synchronize accesses to kernel data structures. Moreover, the Fiasco microkernel is close to RTLinux even though the interrupt handler under RTLinux runs in kernel mode and in the kernel's address space while Fiasco handlers run in their own task.

5.2 Microbenchmarks

We carried out a small series of measurements to evaluate the overhead of our synchronization mechanisms, and to get clues for future optimizations.

We implemented a simple one-word counter and protected its increment operation using the following synchronization schemes: CAS; a wait-free helping lock (Fiasco microkernel's new synchronization); and wait-free object lock with the operation running in a different thread (Fiasco microkernel's old Massalin–Pu single-server-style synchronization). For comparison, we measured an unprotected counter, and a complete address-space-crossing short-IPC operation in the Fiasco microkernel (needs no lock), and we put all results into re-

ness for the Fiasco microkernel; these poor results were caused by a kernel bug that has since been fixed.

System	Cycles [P5]	Cycles [P-II]
counter, unsynchronized	2	2
counter, CAS	13	12
counter, Fiasco thread lock	245	245
counter, old Massalin–Pu–style thread lock (includes one context switch)	627	607
IPC	653	810
IPC, L4/x86	398	438
IPC, L4/x86, small addr. space ^a	184	300

^aL4/x86 offers an optimization called “small address spaces,” which significantly reduces context-switch cost for small address spaces by implementing it using a segment switch instead of a page-table switch [9].

Table 2 Synchronization overhead (under no contention) in the Fiasco microkernel on two different machines. For comparison, we show IPC times (one-way) for the Fiasco microkernel and for L4/x86.

We measured the numbers in the P5 column on a 133-MHz Pentium box, and the number in the P-II column on a 400-MHz Pentium-II box. We used normal C or C++ programs (not hand-optimized assembly) to conduct the measurements.

lation with the performance of Liedtke's L4/x86's IPC performance. Table 2 shows the results.

We are quite satisfied with the performance overhead of our new helping-lock implementation. Even though we are yet to optimize our code, we have already experienced a more-than-twofold improvement in comparison to the implementation of Massalin's and Pu's single-server scheme.

6 Nonblocking synchronization in user-mode programs

In this section we discuss how our design methodology can be applied to multithreaded user-level programs.

Let us recall three preconditions for the effectiveness of our methodology for nonblocking design: First, MW-CAS can only be simulated if concurrent access to the shared data can be disabled. Second, to ensure wait-freedom, critical sections protected by priority-inversion-safe locks must not block. Third, helping only

works if the threading system provides priority inheritance. Meeting these conditions for user-level programs is most definitely possible, but can be difficult. We discuss the conditions in turn.

The interrupt-disabling method to prevent preemptions does not work on user level. Therefore, disabling concurrent access implies some kind of locking. As critical sections accessing data that is updated using simulated MWCAS are typically very short, priority inversion is best prevented by employing preemption-safe locks (i.e., locks that prevent descheduling a lock-holding thread in favor of a thread that shares the lock-protected data structure). In general, the locking mechanism depends on the underlying operating system. For example, spin locks can be used on multiprocessor systems that always gang-schedule all of the program's threads; uniprocessors can use the operating-system-assisted MWCAS implementation we discussed in Section 2.1, or an operating-system-assisted preemption-safe lock.

To avoid blocking inside critical sections, user programs must take extra care typically unnecessary in the kernel: They need to ensure that critical sections do not trigger page faults leading to paging. For that, user programs need operating-system support.

Optimally, the operating system should support priority inheritance in the kernel.

In summary, multi-threaded user programs can use our design technique if the operating system provides some support that real-time systems provide frequently, or can easily implement: MWCAS support, preemption-safe locking, memory pinning, and priority inheritance.

7 Summary and conclusion

We introduced a pragmatic methodology for designing nonblocking real-time systems that is not dependent on an atomic memory-modification primitive like CAS2; just CAS is sufficient.

Our methodology consists of four basic guidelines: (1) partition the system into global and local objects; (2) implement the global objects using lock-free synchronization as far as possible; (3) protect the other objects using locks with priority inheritance; (4) never wait for events inside critical sections. We argued that following these rules ensures wait-freedom.

We derived three conditions for an operating system on which our methodology becomes applicable for wait-free user-mode programs: (1) the operating system must provide help for a user-mode implementation of MWCAS, either directly or by supporting preemption-safe locks; (2) it must provide a service for memory pinning; (3) it must support priority inheritance.

We proposed a wait-free locking-with-helping mechanism with priority inheritance, and we showed that it is similar in effect to, but better performing than, the single-server synchronization mechanism introduced by Massalin and Pu [16]. We devised a monitor implementation that works on top of our locking mechanism.

The application of our methodology can lead to systems with excellent real-time properties: We have built the Fiasco microkernel using the methodology. Together with L⁴Linux, the Fiasco microkernel reaches a level of preemptability that is close to that of RTLinux.

Currently our work has two significant limitations. First, our performance results are preliminary in many ways. Our next steps will be to analyze in more detail what is causing worst-case interrupt latencies of more than 50 µs, and to look at processor dependencies. From this evaluation, we intend to develop a model for predicting worst-case interrupt latencies for our methodology. Second, we have not compared the performance of our synchronization primitives to the performance of primitives found in other commercial and research operating systems. Both limitations are being addressed as part of the first author's thesis work.

In the near future, we plan to add multiprocessor support to the Fiasco microkernel in order to verify our methodology for multiprocessors. Following which, we plan to optimize the Fiasco microkernel's locking-with-helping mechanism and thread switching.

Also, we plan to research the applicability of the techniques for user-level programs in more depth, that is, with real software and measurements.

Availability

The Fiasco microkernel is freely available; researchers are invited to study the implementation of our design methodology, and to experiment with it.

Fiasco and L⁴Linux can be downloaded from <http://os.inf.tu-dresden.de/drops/>.

Acknowledgements

We would like to thank Frank Mehnert for providing his measurement framework, and Michael Peter for improving Fiasco's synchronization primitives. We are grateful to our shepherd, Sheila Harnett, and to our anonymous reviewers for their valuable suggestions.

Special thanks go to Thomas Roche who helped debugging our prose.

This project has been partially funded by the Deutsche Forschungsgemeinschaft in the framework of the Sonderforschungsbereich 358, and supported by generous grants from IBM (University Partnership and Shared University Research programs) and from Intel (MRL Lab Hillsboro).

References

- [1] Atul Adya, Barbara Liskov, Robert Gruber, and Umesh Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of SIGMOD*, San Jose, CA, May 1995.
- [2] James H. Anderson, Srikanth Ramamurthy, and Rohit Jain. Implementing wait-free objects on priority-based systems. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–238, Santa Barbara, California, 21–24 August 1997.
- [3] James H. Anderson, Srikanth Ramamurthy, and Kevin Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions of Computer Systems*, 15(2):134–165, May 1997.
- [4] B. N. Bershad. Practical considerations for non-blocking concurrent objects. In Robert Werner, editor, *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 264–274, Pittsburgh, PA, May 1993. IEEE Computer Society Press.
- [5] Peter A. Buhr and Michael Fortier. Monitor classification. *ACM Computing Surveys*, 27(1):63–107, March 1995.
- [6] Michael Greenwald. *Non-blocking Synchronization and System Design*. PhD thesis, Stanford University, August 1999.
- [7] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, October 28–31, 1996. Seattle, WA, pages 123–136, Berkeley, CA, USA, October 1996. USENIX.
- [8] H. Härtig, R. Baumgartl, M. Borris, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.
- [9] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ -kernel-based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, pages 66–77, Saint-Malo, France, October 1997.
- [10] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Using threads in interactive systems: A case study. In *14th ACM Symposium on Operating System Principles (SOSP)*, pages 94–105, Asheville, NC, December 1993.
- [11] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [12] Michael Hohmuth. The Fiasco kernel: Requirements definition. Technical Report TUD-FI-12, TU Dresden, December 1998. Available from URL: http://os.inf.tu-dresden.de/papers_ps/fiasco-spec.ps.gz.
- [13] John H. Howard. Signaling in monitors. In *Second International Conference on Software Engineering*, pages 47–52, San Francisco, CA, October 1976.
- [14] B. W. Lampson and D. D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.
- [15] J. Liedtke. On μ -kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, pages 237–250, Copper Mountain Resort, CO, December 1995.
- [16] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.

- [17] Frank Mehnert. L4RTL: Porting RTLinux API to L4/Fiasco. In *Workshop on a Common Microkernel System Platform*, Kiawah Island, SC, December 1999.
- [18] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96)*, pages 267–275, New York, USA, May 1996. ACM.
- [19] M. Moir. Transparent support for wait-free transactions. *Lecture Notes in Computer Science*, 1320:305, 1997.
- [20] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.
- [21] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.
- [22] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, 2–23 August 1995. Erratum available at <ftp://ftp.cs.rpi.edu/pub/valoisj/podc95-errata.ps.gz>.
- [23] John D. Valois. *Lock-Free Data Structures*. PhD thesis, Rensselaer Polytechnic Institute, May 1995.
- [24] Victor Yodaiken and Michael Barabanov. A Real-Time Linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*, Anaheim, CA, January 1997. The USENIX Association.

Scalability of Linux Event-Dispatch Mechanisms

Abhishek Chandra

*Department of Computer Science,
University of Massachusetts, Amherst
abhishek@cs.umass.edu*

David Mosberger

*Hewlett Packard Laboratories,
Palo Alto, CA
davidm@hpl.hp.com*

Abstract

Many Internet servers these days have to handle not just heavy request loads, but also increasingly face large numbers of concurrent connections. In this paper, we discuss some of the event-dispatch mechanisms used by Internet servers to handle the network I/O generated by these request loads. We focus on the mechanisms supported by the Linux kernel, and measure their performance in terms of their dispatch overhead and dispatch throughput. Our comparative studies show that POSIX.4 Real Time signals (RT signals) are a highly efficient mechanism in terms of the overhead and also provide good throughput compared to mechanisms like select() and /dev/poll. We also look at some limitations of RT signals and propose an enhancement to the default RT signal implementation which we call signal-per-fd. This enhancement has the advantage of significantly reducing the complexity of a server implementation, increasing its robustness under high load, and also potentially increasing its throughput. In addition, our results also show that, contrary to conventional wisdom, even a select() based server can provide high throughput, even though it has high overhead, if its overhead is amortized by performing more useful work per select() call.

1 Introduction

The fast growth of the Web and e-commerce has led to a large increase in Internet traffic. Most network applications such as Web servers and proxies have to handle heavy loads from clients spread all across the globe. In addition to high request rates, servers also have to handle a large number of concurrent connections, many of which are idle most of the time. This is because the connection times are large due to (i) the “last-mile problem” [3], which has the effect that most clients connect to the Internet through slow modems, and (ii) due to the geographically distributed nature of the Internet, which causes much of the traffic to travel across many hops, in-

creasing both latency and the probability of packet drops due to congestion. For Web servers, the problem of long connections is exacerbated by the HTTP/1.1 protocol [5], which provides for persistent TCP connections that can be reused to handle multiple interactions with the server. These persistent connections further add to the length of the connection times. The bottom line is that servers need to service the high incoming request load, while simultaneously handling a large number of concurrent connections efficiently.

To handle these demands, many high-performance Web servers are structured as event-handling applications [9, 16, 18]. These servers employ event-dispatch mechanisms provided by the underlying operating system to handle the network I/O on multiple concurrent connections. Some studies have looked at the scalability issues of such mechanisms and found that traditional dispatch mechanisms are not very scalable [1]. While the performance of Web servers clearly is important, we should not forget that there are many other Internet services, such as ftp servers, proxy caches, and mail servers, that have to deal with similar scalability concerns. For example, poor scalability is one of the primary reasons the number of concurrent connections on many ftp servers is limited to a small number (around 30-50)¹.

Another approach to building Internet servers that can handle high request loads and large number of concurrent connections is to move the entire application into kernel space. Recent efforts in this direction have produced dramatic results for Web servers (e.g., TUX [17]). However, this does not obviate the need for efficient event-dispatch mechanisms. In fact, it is our contention that due to security and robustness concerns, many server sites are likely to prefer running Internet servers in user space, provided that they can achieve performance that is comparable to a kernel space solution. Efficient event dispatch mechanisms are also essential

¹The scalability problems of the most popular ftp servers is partially due to the fact that they are using a process-per-connection model. This could be fixed by using an event-oriented model, though security considerations may not always permit adopting such a model.

for those applications that may be important for some sites (e.g., ftp), but perhaps not quite important enough to warrant the effort of developing an OS-specific kernel solution.

In this paper, we look at the different Linux event-dispatch mechanisms used by servers for doing network I/O. We try to identify the potential bottlenecks in each case, with an emphasis on the scalability of each mechanism and its performance under high load. We use two metrics to determine the efficiency of each mechanism, namely, the event-dispatch overhead and the dispatch throughput. The mechanisms we study in particular are the `select()` system call, `/dev/poll` interface and POSIX.4 Real Time signals (RT signals), each of which is described in more detail in the following sections. Our studies show that RT signals are an efficient and scalable mechanism for handling high loads, but have some potential limitations. We propose an enhancement to the kernel implementation of RT signals that overcomes some of these drawbacks, and allows for robust performance even under high load. We also measure the performance of a variant of the `select()` based server which amortizes the cost of each `select()` call, and show that it is more scalable in terms of the server throughput.

The rest of the paper is organized as follows. In Section 2, we describe the primary event-dispatch mechanisms supported by the Linux kernel, and discuss some of the previous work in this regard. In Section 3, we compare some of these mechanisms for their dispatch overhead. We discuss RT signals in more detail, identifying their limitations and propose an enhancement to the default implementation of RT signals in the Linux kernel. In Section 4, we present a comparative study of some of the mechanisms from the perspective of throughput achieved under high loads. Finally, we present our conclusions in Section 5.

2 Event-Dispatch Mechanisms

In this section, we first discuss the two main schemes employed by servers for handling multiple connections. Next, we look at the various event-dispatch mechanisms supported by the Linux kernel that can be employed by Web servers for doing network I/O. We follow this up with a discussion of previous work that has focussed on the scalability of some of these mechanisms, including other mechanisms that have been proposed to overcome some of their drawbacks.

2.1 Handling Multiple Connections

There are two main methodologies that could be adopted by servers for performing network I/O on multiple concurrent connections.

- *Thread-based:* One way to handle multiple connections is to have a master thread accepting new connections, that hands off the work for each connection to a separate service thread. Each of these service threads is then responsible for performing the network I/O on its connection. These service threads can be spawned in two ways:
 - *On-demand:* Each service thread is forked whenever a new connection is accepted, and it then handles the requests for the connection. This can lead to large forking overhead under high load when there are large number of new connections being established.
 - *Pre-forked:* The server could have a pool of pre-forked service threads. Whenever the master thread receives a new connection, it can hand over the connection to one of the threads from the pool. This method prevents the forking overhead, but may require high memory usage even under low loads.
- *Event-based:* In an event-based application, a single thread of execution uses non-blocking I/O to multiplex its service across multiple connections. The OS uses some form of event notification to inform the application when one or more connections require service. For this to work, the application has to specify to the OS the set of connections (or, more accurately, the set of file-descriptors) in which it is interested (*interest set*). The OS then watches over the interest set and whenever there's activity on any of these connections, it notifies the application by dispatching an event to it. Depending on the exact event-dispatch mechanism in use, the OS could group multiple notifications together or send individual notifications. On receiving the events, the server thread can then handle the I/O on the relevant connections.

In general, thread-per-connection servers have the drawback of large forking and context-switching overhead. In addition, the memory usage due to threads' individual stack space can become huge for handling large number of concurrent connections. The problem is even more pronounced if the operating system does not support kernel-level threads, and the application has to use processes or user-level threads. It has been shown that thread-based servers do not scale well at high loads [7].

Hence, many servers are structured as event-based applications, whose performance is determined by the efficiency of event notification mechanisms they employ. Pure event-based servers do not scale to multiprocessor machines, and hence, on SMP machines, hybrid schemes need to be employed, where we have a multi-threaded server with each thread using event-handling as a mechanism for servicing concurrent connections. Even with a hybrid server, the performance of event-based mechanisms is an important issue. Since efficient event dispatching is at the core of both event-based and hybrid servers, we will focus on the former here.

2.2 Linux Kernel Mechanisms

As described above, event-based servers employ event-dispatch mechanisms provided by the underlying operating system to perform network I/O. In this section, we describe the mechanisms supported by the Linux kernel for event notification to such applications. Following are the mechanisms supported by the Linux kernel.

- ***select() system call:*** `select()` [15] allows a single thread or process to multiplex its time between a number of concurrently open connections. The server provides a set of file-descriptors to the `select()` call in the form of an `fdset`, that describes the interest set of the server. The call returns the set of file-descriptors that are ready to be serviced (for read/write, etc.). This ready set is also returned by the kernel in the form of an `fdset`.

The main attributes of the `select()` based approach are:

- The application has to specify the interest set repeatedly to the kernel.
- The interest set specification could be sparse depending on the descriptors in the set, and could lead to excess user-kernel space copying. The same applies when returning the ready set.
- The kernel has to do a potentially expensive scan of the interest set to identify the ready file descriptors.
- If the kernel wakes up multiple threads interested in the same file descriptor, there could be a *thundering herd* problem, as multiple threads could vie for I/O on the same descriptor. This, however, is not a problem with Linux 2.4.0 kernel, as it supports single thread wake-up.
- ***poll() system call:*** `poll()` [15] is a system call identical to `select()` in its functionality, but uses

```
// Accept a new connection
int sd = accept(...);

// Associate an RT signal
// with the new socket
fcntl(sd, F_SETOWN, getpid());
fcntl(sd, F_SETSIG, SIGRTMIN);

// Make the socket non-
// blocking and asynchronous
fcntl(sd, F_SETFL, O_NONBLOCK|O_ASYNC);
```

Figure 1: Associating a new connection with an RT signal

a slightly different interface. Instead of using an `fdset` to describe the interest set, the server uses a list of `pollfd` structures. The kernel then returns the set of ready descriptors also as a list of `pollfd` structures. In general, `poll()` has a smaller overhead than `select()` if the interest set or ready set is sparse. But if these sets are dense, then the overhead is usually higher because `pollfd` structures are bigger than a bit (they are a few bytes typically). Other than that, `poll()` has the same problems as `select()`.

- ***POSIX.4 Real Time Signals:*** POSIX.4 Real Time signals (RT signals) [6] are a class of signals supported by the Linux kernel which overcome some of the limitations of traditional UNIX signals. First of all, RT signals can be queued to a process by the kernel, instead of setting bits in a signal mask as is done for the traditional UNIX signals. This allows multiple signals of the same type to be delivered to a process. In addition, each signal carries a `siginfo` payload which provides the process with the context in which the signal was raised.

A server application can employ RT signals as an event notification mechanism in the following manner. As shown in figure 1, the server application can associate an RT signal with the socket descriptors corresponding to client connections using a series of `fcntl()` system calls. This enables the kernel to enqueue signals for events like connections becoming readable/writable, new connection arrivals, connection closures, etc. Figure 2 illustrates how the application can use these signal notifications from the kernel to perform network I/O. The application can block the RT signal associated with these events (`SIGRTMIN` in figure 2) and use `sigwaitinfo()` system call to synchronously dequeue the signals at its convenience. Using `sigwaitinfo()` obviates the need for asynchronous

```

sigset_t signals;
siginfo_t siginfo;
int signum, sd;

// Block the RT signal
sigemptyset(&signals);
sigaddset(&signals, SIGRTMIN);
sigprocmask(SIG_BLOCK, &signals, 0);

while (1) {
    // Dequeue a signal from the signal queue
    signum = sigwaitinfo(&signals, &siginfo);

    // Check if the signal is an RT signal
    if (signum == SIGRTMIN) {
        // Identify the socket associated with the signal
        sd = siginfo.si_fd;
        handle(sd);
    }
}

```

Figure 2: Using RT signals for doing network I/O

signal delivery and saves the overhead of invoking a signal handler. Once it fetches a signal, the *siginfo* signal payload enables the application to identify the socket descriptor for which the signal was queued. The application can then perform the appropriate action on the socket.

One problem with RT signals is that the signal queue is finite, and hence, once the signal queue overflows, a server using RT signals has to fall back on a different dispatch mechanism (such as *select()* or *poll()*). Also, *sigwaitinfo()* allows the application to dequeue only one signal at a time. We'll talk more about these problems in section 3.3.

Event-dispatching mechanisms also exist in operating systems other than Linux. For instance, Windows NT provides *I/O completion ports* [12], which are primarily used for thread-based servers. With I/O completion ports, there is a single event queue and a fixed number of pre-forked threads which are used to process the events. There is a throttling mechanism to ensure that at most N threads are running at any given time. This makes it possible to pre-fork a relatively large number of threads while avoiding excessive context switching during busy periods. Even though primarily intended for thread-based servers, it should be possible to use I/O ports in conjunction with asynchronous I/O to implement hybrid servers.

2.3 Previous Work

Banga et al. [1] have studied the limitations of a *select()* based server on DEC UNIX, and shown the problems with its scalability, some of which we have discussed above. They have proposed a new API in [2], which allows an application to specify its interest set incrementally to the kernel and supports event notifications on descriptors instead of state notifications (as in the case of *select()* and *poll()*). The system calls proposed as part of this API are *declare_interest()*, which allows an application to declare its interest in a particular descriptor, and *get_next_event()*, which is used to get the next pending event(s) from the kernel.

Another event-dispatch mechanism is the */dev/poll* interface, which is supported by the Solaris 8 kernel [14]. This interface is an optimization for the *poll()* system call. Recently, Provos et al. [10] have implemented the */dev/poll* interface in the Linux kernel. This interface works as follows. The application first does an *open()* on the */dev/poll* device, which creates a new interest set for the application. From this point onwards, the application can add a new socket to this interest set incrementally by creating a *pollfd* struct and writing it to the */dev/poll* device. Finally, the polling is done by using an *ioctl()* call, which returns a list of *pollfd* structs corresponding to the set of ready descriptors. Further, the overhead of user-kernel copies can be reduced by using *mmap()* to map the array of *pollfd* structs onto the */dev/poll* device. In [10], the

`/dev/poll` interface is shown to be an improvement on the traditional `poll()` implementation, especially as it reduces the cost of specifying the interest set to the kernel. Hence, in our experiments, we have used `/dev/poll` instead of `poll()` for comparison to other dispatch mechanisms.

RT signals have been used for network I/O in the *ph-htpd* [4] Web server. Provos et al. have discussed its implementation and some of its shortcomings, such as the potential of signal queue overflow and the ability of `sigwaitinfo()` system call to fetch only one signal at a time. They have proposed a new system call `sigtimedwait4()` which allows the server to dequeue multiple signals from the signal queue [11].

3 Dispatch Overhead

In this section, we look at the first scalability parameter for event-dispatch mechanisms, namely the overhead involved in handling requests as a function of the number of concurrent connections. This parameter becomes important in the context of large number of idle or slow connections, irrespective of the actual active load on the server. In what follows, we first present an experimental study of some of the Linux dispatch mechanisms, and then discuss some of the insights from this study. We follow this up with a detailed discussion of RT signal behavior, including their limitations. We then propose an enhancement to the implementation of RT signals in the Linux kernel to overcome some of these limitations.

3.1 Comparative Study

In this section, we present the results of our comparative study of some of the kernel mechanisms discussed above. The main goal of this study is to look at the behavior of Web servers under high load in terms of their CPU usage as the number of concurrent connections (most of them idle) increases.

3.1.1 Experimental Testbed

To conduct the experimental study, we implemented a set of *micro Web servers* (μ servers), each employing a different event-dispatch mechanism. Most of the request handling and administrative code in these μ servers is identical to avoid differences in performance arising due to other factors. This ensures that the different versions are as similar as possible. Also, using the μ servers instead of widely-used, full-fledged Web servers allows us to focus on the performance impact of the dispatch mechanisms by reducing all other overheads to the absolute minimum. Moreover, existing Web servers have an

underlying event-handling architecture (such as process-per-connection for Apache), which may not be suitable for the purpose of our study. Thus, the μ servers do very simple HTTP protocol processing, and the various μ servers differ only in their use of the event-dispatch mechanism. Specifically, we compared μ servers employing `select()`, `/dev/poll` and RT signals as their event-dispatch mechanisms. While this approach of using the μ servers does not answer the question of how important the event-dispatch costs are as part of the overall server overhead for commercial Web servers, it does help in determining the limit on the scalability of such servers. This is because, even if the dispatch overhead is tiny with a small number of connections, non-linear scaling behavior could magnify this overhead with increasing number of connections until it eventually becomes the first order bottleneck.

Each of these μ servers was run on a 400 MHz Pentium-III based dual-processor HP NetServer LPr machine running Linux 2.4.0-test7 in uniprocessor mode. The client load was generated by running *httperf* [8] on ten B180 PA-RISC machines running HP-UX 11.0. The clients and the server were connected via a 100 Mbps Fast Ethernet switch. To simulate large number of concurrent and idle connections, each *httperf* was used to establish a set of persistent connections, each of which generated periodic requests to the μ server. The effect was that at all times, some of the connections were active while the rest were idle, and these active and idle connection sets kept changing with time. Thus, in these experiments, the connection rate was different from the request rate (with each connection generating multiple requests). The server's reply size was 92 bytes. In each experiment, the total request rate was kept constant, while the number of concurrent connections was varied to see the effect of large number of idle connections on server performance.

To measure the CPU usage of the μ server, we inserted an `idle_counter` in the kernel running the μ server. This `idle_counter` counted the idle cycles on the CPU. We computed the CPU load imposed by the μ server by comparing the idle cycles for the system with and without the μ server running on it. The server reply rate and response times were measured by the *httperf* clients.

3.1.2 Experimental Results

As part of our comparative study, we ran experiments to measure the performance of three μ servers based on `select()`, `/dev/poll` and RT signals respectively. In each experiment, the clients were used to generate a fixed request rate, and the number of concurrent connections was increased from 250 to 3000. Figure 3 shows

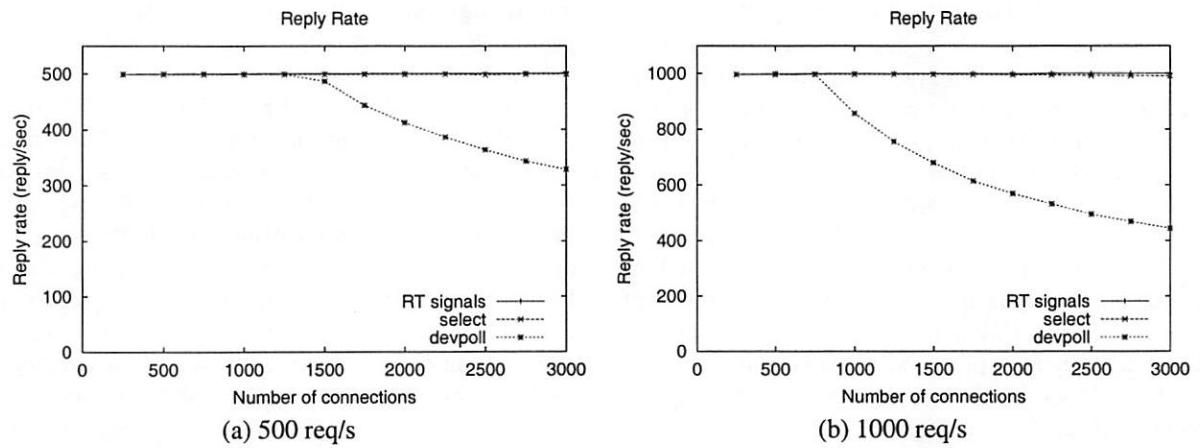


Figure 3: Reply rate with varying number of concurrent connections

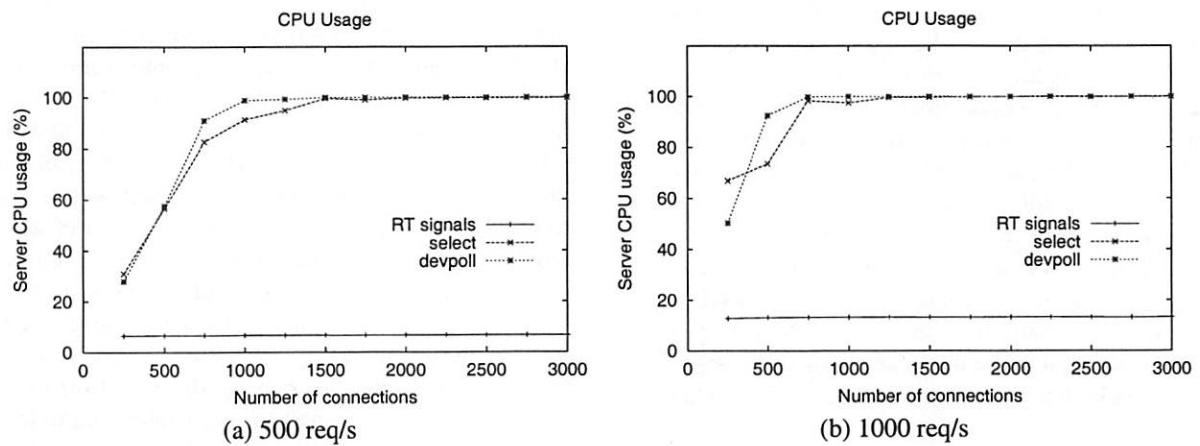


Figure 4: CPU usage with varying number of concurrent connections

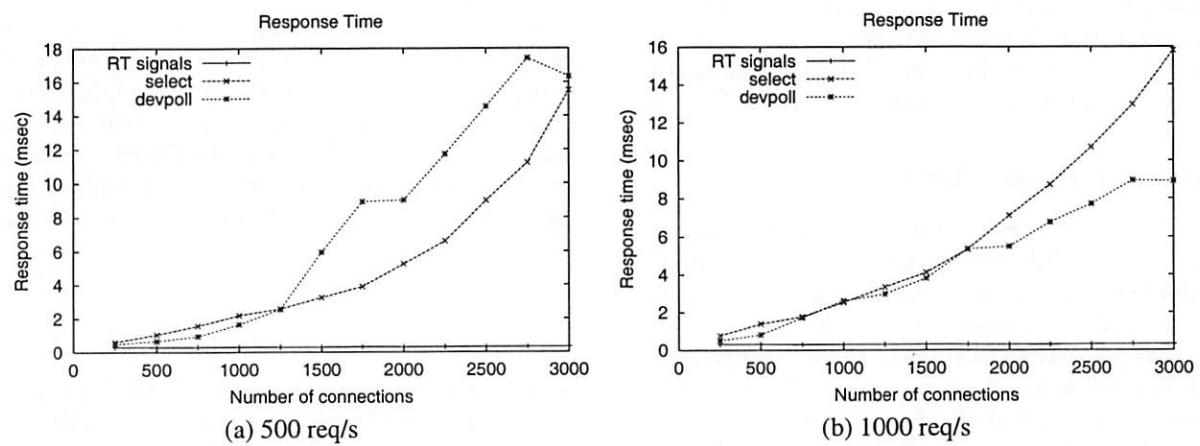


Figure 5: Response time with varying number of concurrent connections

the reply rates achieved by the servers for request rates of 500 req/s and 1000 req/s respectively. As can be seen from the figure, the reply rate matches the request rate for the RT signal and `select()` based servers at all points. On the other hand, the reply rate starts dropping off for the `/dev/poll` based server after a point. This is because the server becomes overloaded and starts dropping connections beyond a certain load. We cannot explain why the overload behavior of `/dev/poll` is so bad.

The more interesting figures are figures 4 and 5, which show the CPU usage and the average response time respectively for each of the μ servers, as the number of concurrent connections is increased. As can be seen from figure 4, the CPU usage for both `select()` and `/dev/poll` increases with the number of concurrent connections and they become saturated after a certain point. On the other hand, the CPU usage for RT signals is insensitive to the number of idle connections. The RT signal based server's CPU usage is about 6.67% on average for the 500 req/s case, while it is about 13.25% for the 1000 req/s case. Thus, the CPU overhead of RT signals seems to be dependent only on the request rate. Also, the RT signal CPU usage is dramatically lower than either `select()` or `/dev/poll` based servers. A similar behavior is seen for the response time in figure 5. Once again, the response time increases for both the `select()` and `/dev/poll` based servers with the number of connections. On the other hand, the RT signal based server shows a very small response time for each of the request rates (about 0.3 ms in each case). Further, this response time is independent of the number of concurrent connections. Note that, even though the absolute value of the response times in the graphs may not seem significant from the perspective of an end user, it is the shape of these graphs which is significant, as these curves reflect the scalability of the dispatch mechanisms.

Thus, the results in this section show that RT signals have very small dispatch overhead and also that this overhead does not depend on the number of concurrent or idle connections being handled by the server. Rather, it is determined only by the active work being done by the server.

3.2 RT Signals: Reasons for Efficiency

From our comparative study, we observe that RT signals have a relatively low overhead compared to `select()` and `/dev/poll` event-dispatch mechanisms. Further, this overhead seems to be independent of the number of idle connections, and depends only on the active request rate. In other words, RT signals show essentially ideal behavior. In this section, we discuss the reasons for the better performance of RT signals in more

detail.

RT signals are more efficient due to the following reasons:

- First, the server only needs to specify its interest set to the kernel incrementally. This is because the server application associates an RT signal with each socket file descriptor at the time of its creation (just after the `accept()` system call). From this point onwards, the kernel automatically generates signals corresponding to events on the descriptor, and thus obviates the need for the application to specify its interest in the descriptor again and again (as is the case with `select()` system call). This functionality is similar to the `declare_interest()` API proposed in [2].
- Unlike `select()`, `poll()` and `/dev/poll`, in the case of RT signals, the kernel does not know about the interest set explicitly. Rather, whenever there's an event on one of the descriptors, the kernel enqueues a signal corresponding to the event without having to worry about the interest set. Thus, the interest set is totally transparent to the kernel and this gets rid of the overhead of scanning each descriptor in the interest set for activity on every polling request from the application.
- Based on the `fd` field in the signal payload, the application can identify the active descriptor immediately without having to potentially check each descriptor in the interest set (as in the case of `select()`).
- By blocking the relevant RT signal and using `sigwaitinfo()` for dequeuing signals from the signal queue, the overhead of calling a signal handler is avoided.

3.3 Limitations of RT signals

In spite of their efficiency, RT signals, as currently implemented in Linux, have some potential limitations. These limitations arise from the fact that the signal queue is a limited resource. Since each event results in a signal being appended to the signal queue, a few active connections could dominate the signal queue usage or even trigger an overflow. The former could result in unfair service and the latter could cause a deadlock-like situation in which the server can no longer make any progress, and appears to be suspended or hung.

To understand how a signal queue overflow can lead to a “hung” server, note that once the queue is full, no further signals can be enqueued and hence all future events are dropped. Of course, eventually the server would drain the queue and new events would start to

come in again. However, those events that got dropped are lost forever. Further, notice that the signal queue is delinked from the TCP buffers and there is no feedback mechanism between the two. Thus, even after the signal queue fills up and starts losing signals, there is nothing to throttle the TCP traffic. Thus, even though events are occurring on the open connections and the listening socket, the server loses notifications corresponding to these events. In other words, there is a “notification leak” at the server. If one of the lost events happened to indicate, for example, that the listen queue has pending connections, the server may never realize that it ought to call `accept()` to service those connections. Similarly, if an event got dropped that indicated that a particular connection is now readable, the server may never realize that it should call `read()` on that connection. Over time, the more events are dropped, the more likely it becomes that either some connections end up in a suspended state or that the listening socket is no longer serviced. In either case, throughput will suffer and eventually drop to zero.

To avoid this kind of suspended state, the Linux kernel sends a `SIGIO` signal to the application when the signal queue overflows. At this point, the application can recover from the overflow by falling back onto some other event dispatch mechanism. For example, the application could use `select()` or `poll()` to detect any events that may have been dropped from the signal queue. Unfortunately, using a fallback mechanism comes with its own set of problems. Specifically, there are two issues:

- First, having to handle signal queue overflows by switching onto another mechanism makes the application complex. It may require translating the interest set from the (implicit) form used by the RT signal mechanism into the explicit form used by the other mechanism (e.g.: setting up the fdsets for `select()` or `pollfd` lists for `poll()`, etc.). Furthermore, the application has to receive and service the kernel notifications in a different manner. Also, this transition needs to be done very carefully, as losing even a single event could potentially create the suspended state situation mentioned above.
- Second, switching over to a non-scalable mechanism also has the potential to make the application sluggish. Since the application is already under overload (which led to the signal queue overflow in the first place), using a high-overhead mechanism for recovery could overload the server even further, potentially sending it into a tailspin.

Another drawback with RT signals is that each call to `sigwaitinfo()` dequeues exactly one signal from

the queue. It cannot return multiple events simultaneously, leading to high number of system calls to retrieve multiple events, which might be a problem under high load.

Thus, using RT signals as implemented in the kernel has some potential drawbacks even if they are used in conjunction with another mechanism.

3.4 Signal-per-fd: RT Signal Enhancement

As discussed above, having to handle a signal queue overflow could be potentially costly as well as complex for an application. It would be desirable, therefore, if signal queue overflows could be avoided altogether. To understand why signal queue overflows are happening in the first place, note that there’s a potential of multiple events being generated for each connection, and hence multiple signals being enqueued for each descriptor. But, most of the time, the application does not need to receive multiple events for the same descriptor. This is because even when an application picks up a signal corresponding to an event, it still needs to check the status of the descriptor for its current state, as the signal might have been enqueued much before the application picks it up. In the meantime, it is possible that there might have been other events and the status of the descriptor might have changed. For instance, the application might pick up a signal corresponding to a read event on a descriptor *after* the descriptor was closed, so that the application would have to decide what to do with the event in this case. Thus, it might be more efficient and useful if the kernel could coalesce multiple events and present them as a single notification to the application. The application could then check the status of the descriptor and figure out what needs to be done accordingly.

To understand what kind of events can be coalesced together, we have to understand what kind of information the events are supplying to the application. In general, events are coalescable under two scenarios:

- If information from multiple events can be combined to provide consistent information to the application, then such events can be coalesced. This scenario occurs in many GUI systems, such as the X window system [13]. In such systems, typical implementations try to compact motion events in order to minimize system overhead and memory usage in device drivers. For instance, if we receive an event indicating that the mouse moved by (x_1, y_1) , followed by an event that it moved by (x_2, y_2) , then the two events can be merged into a single one indicating that the mouse moved by $(x_1 + x_2, y_1 + y_2)$.
- If events are used only as hints which the appli-

cation might pick up at its leisure (or ignore) and are used only as feedback to guide it in its actions, then such events may be coalesced. This is the scenario in our RT signal-based server, where the signals are being used as hints to notify the server of connection-related events, but the onus is still on the server to verify the connection status and act accordingly.

We propose an enhancement to achieve this coalescing, which we call *signal-per-fd*. The basic idea here is to enqueue a single signal for each descriptor. Thus, whenever there's a new event on a connection, the kernel first checks if there's already a signal enqueued for the corresponding file descriptor, and if so, it does not add a new signal to the queue. A new signal is added for a descriptor only if it does not already have an enqueued signal.

To efficiently check for the existence of a signal corresponding to a descriptor, we maintain a bitmap per process. In this bitmap, each bit corresponds to a file-descriptor and the bit is set whenever there is an enqueued signal for the corresponding descriptor. Note that checking the bit corresponding to a descriptor obviates the need to scan the signal queue for a signal corresponding to the descriptor, and hence, this check can be done in constant time. This bit is set whenever the kernel enqueues a new signal for the descriptor and it is cleared whenever the application dequeues the signal.

By ensuring that one signal is delivered to the application for each descriptor, the kernel coalesces multiple events for a connection into a single notification, and the application then checks the status of the corresponding descriptor for the action to be taken. Thus, if the size of the signal queue (and hence the bitmap) is as large as the file descriptor set size, we can ensure that there would never be a signal queue overflow.

This enhancement has the following advantages:

- Signal-per-fd reduces the complexity of the application by obviating the need to fall back on an alternative mechanism to recover from signal queue overflows. This means that the application does not have to re-initialize the state information for the interest set, etc. that may be required by the second mechanism. In addition, the application does not have to deal with the complexity associated with ensuring that no event is lost on signal queue overflow. Finally, the server would not have to pay the penalty of potentially costly dispatch mechanisms.
- Signal-per-fd also ensures fair allocation of the signal queue resource. It prevents overloaded and misbehaving connections from monopolizing the signal queue, and thus achieves a solution for the proper resource management of the signal queue.

- By coalescing multiple events into a single notification, this mechanism prevents the kernel from providing too fine-grained event notifications to the application, especially as the application might not pick up the notifications immediately after the events. This enhancement thus notifies the application that there *were* events on a descriptor, instead of *how many* events there were. The latter information is often useless to the application as it has to anyway figure out *what* the events were and what the status of the descriptor is.

There are some similarities between the signal-per-fd mechanism and the API proposed in [2] in terms of their functionality. In particular, signal-per-fd allows the kernel to coalesce multiple events corresponding to a descriptor just like the implementation described there. But, signal-per-fd differs from this API in some important ways. First of all, signal-per-fd does not require any change in the system call interface or the kernel API, and is totally transparent to the application. Secondly, the goal of the signal-per-fd mechanism is not only to potentially improve the performance of an RT signal-based server, but also to reduce its complexity, by obviating the need for employing fallback mechanisms.

On the whole, signal-per-fd is a simple enhancement to the implementation of RT signals that can overcome some of their limitations in the context of using them as an event-dispatch mechanism for doing network I/O.

4 Dispatch Throughput

In this section, we look at another parameter associated with the efficiency of event-dispatch mechanisms, namely, the throughput that can be achieved as a function of the active load on the server. This metric is orthogonal to the overhead discussed in the previous section, as this refers to the actual amount of useful work being performed by the server. In what follows, we first provide a comparative experimental study of some of the Linux dispatch mechanisms, including the *signal-per-fd* optimization proposed in the previous section. In addition, we also look at the throughput achieved by a `select()` based server with a minor modification which allows the server to do multiple `accept()`s each time the listening socket becomes ready. Then, we discuss the results of this study and provide some insights into the behavior of the various mechanisms.

4.1 Experimental Study

Here, we experimentally evaluate the throughput achieved by various event-dispatch mechanisms under high load. Our experimental setup is the same as

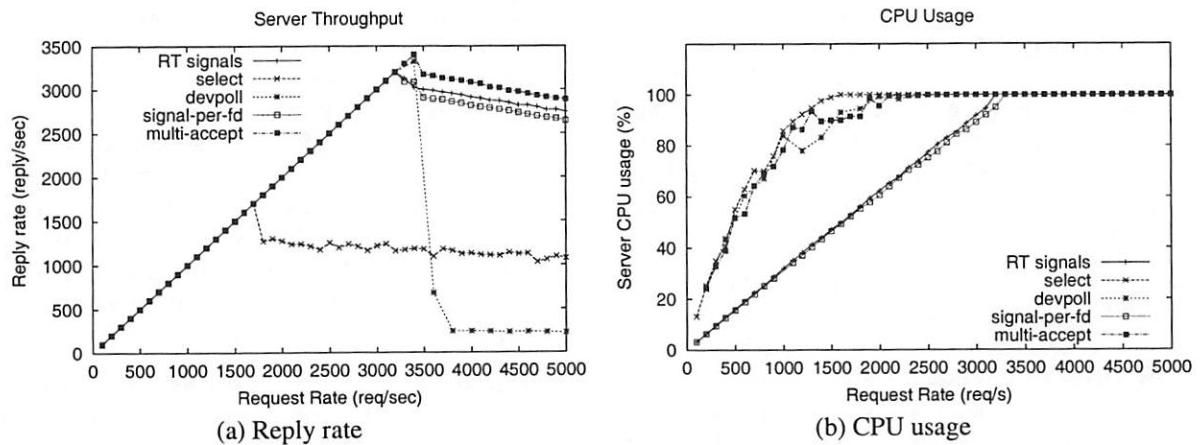


Figure 6: Server performance with 252 idle connections (1B reply size)

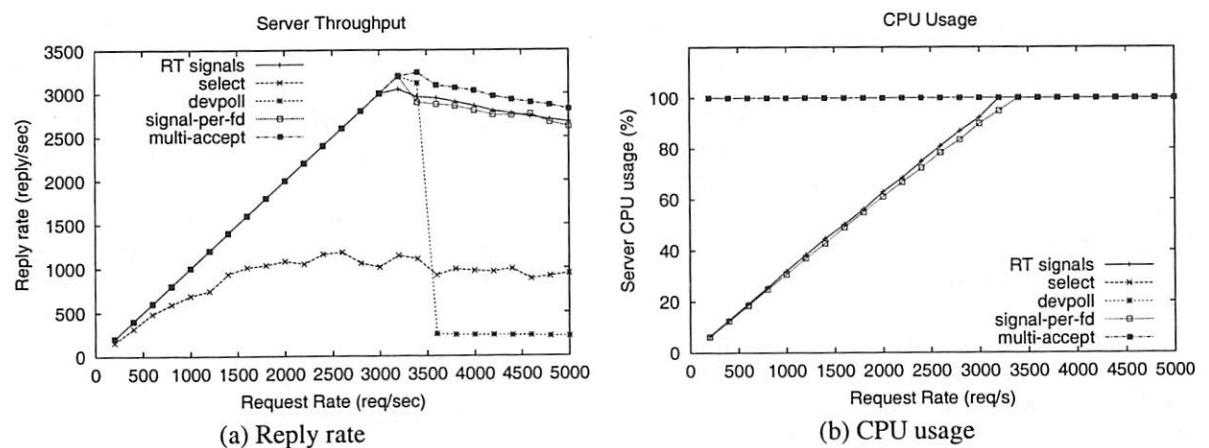


Figure 7: Server performance with 6000 idle connections (1B reply size)

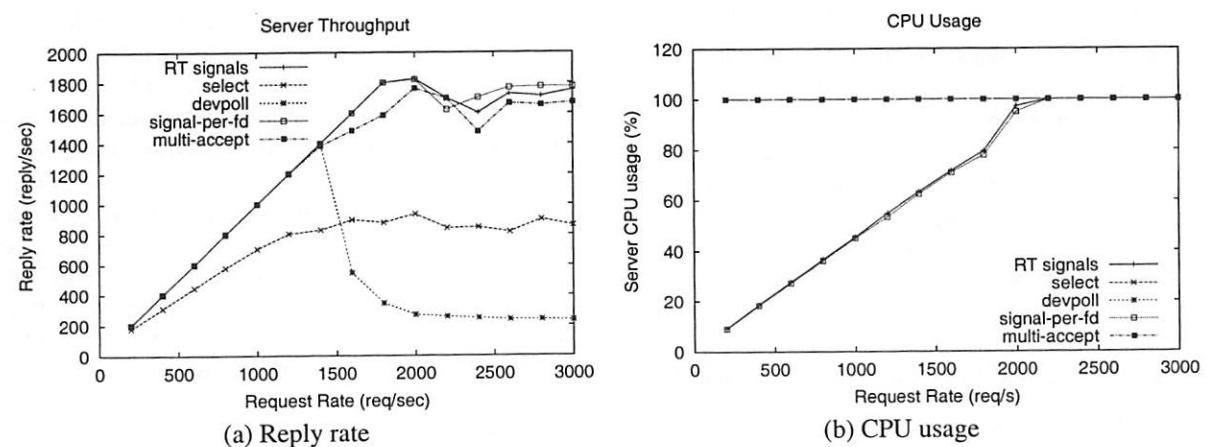


Figure 8: Server performance with 6000 idle connections (6K reply size)

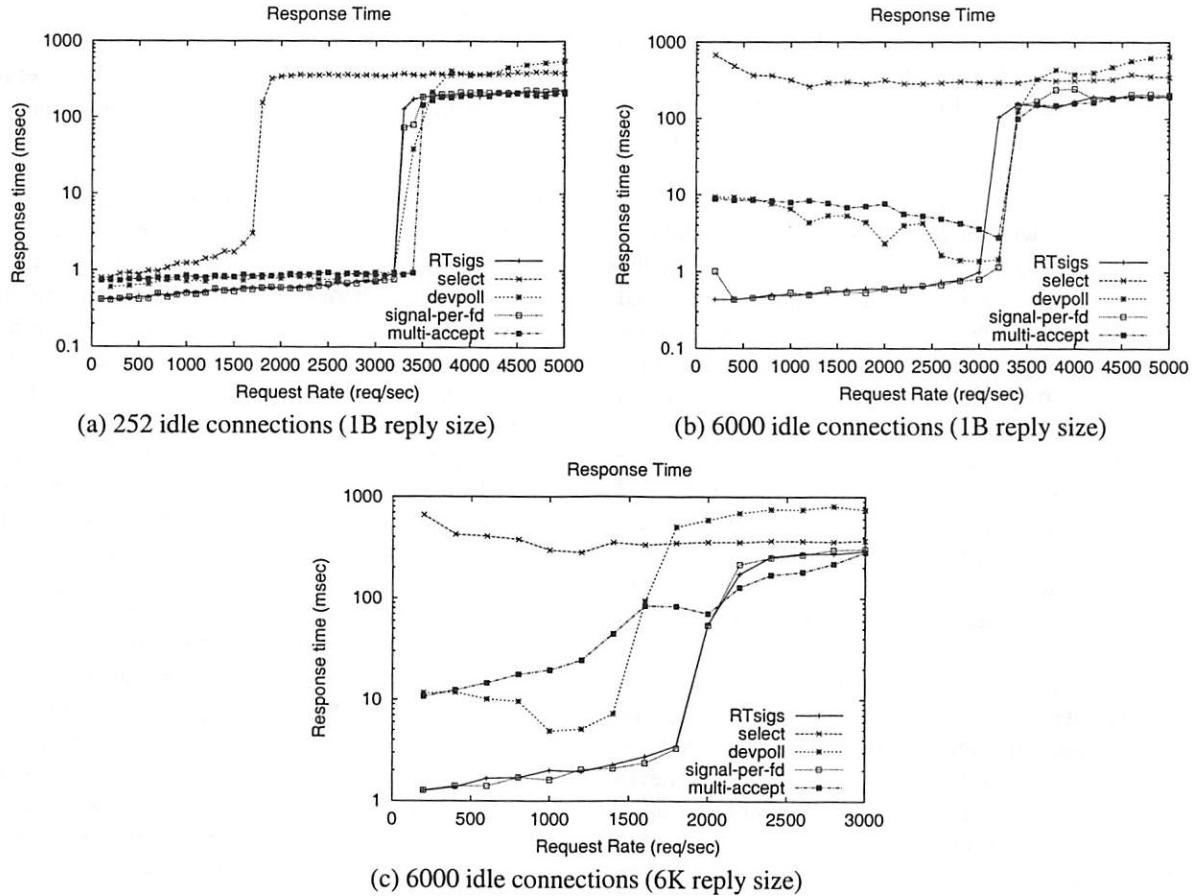


Figure 9: Response time with increasing load

that used in Section 3.1 for comparative study of `select()`, `/dev/poll` and RT signals. In this study, we evaluate two new mechanisms/enhancements as well:

- The signal-per-fd enhancement to RT signals. We have implemented this enhancement in the Linux 2.4.0-test7 kernel, and we ran the RT signal based μ server on the modified kernel for measuring the effect of signal-per-fd.
- A `select()` based μ server which does multiple `accept()`s each time the listening socket becomes ready, as opposed to the standard `select()` based μ server which does only one `accept()` for such a case. We'll refer to this modification as *multi-accept select*. The idea here is to enable the server to identify more connections and perform more useful work per `select()` call.

In order to measure the throughput of these various mechanisms under varying loads, we used a set of idle connections along with a set of *httperf* clients generating requests at high rates. In these experiments, we kept the

number of idle connections fixed for each experiment and varied the request rate.

Figure 6 shows the performance of the servers with 252 idle connections and 1 byte server reply sizes, while figures 7 and 8 plot the same information for 6000 idle connections with server reply sizes of 1 byte and 6KB respectively. As can be seen from figures 6(a), 7(a) and 8(a), the throughput with `select()` plateaus out much before it does for the RT signals (both the default and the signal-per-fd implementations). The fall in reply rate of `/dev/poll` is much more dramatic and again, it seems to perform very poorly under overload. The interesting observation is that multi-accept `select` is able to sustain a high throughput, similar to the RT signals, and even manages to achieve a slightly higher peak throughput in the first two cases. Figures 6(b), 7(b) and 8(b) show the CPU usage for the μ servers. Again, as can be seen from these figures, the CPU usage for RT signals is much less than that for `select()`, multi-accept `select` and `/dev/poll` in all cases, and RT signals reach saturation at a much higher load. In fact, for 6000

idle connections (figures 7(b) and 8(b)), CPU usage is 100% for the `select()`, multi-accept `select` and `/dev/poll` based servers right from the beginning, which can be attributed to their high overhead in handling large number of concurrent connections. On the other hand, the CPU overhead for the RT signals based server (for both the default and signal-per-fd cases) increases linearly with the load in either case. An interesting point to be noted from these figures is that, for the 1 byte reply sizes, the server with the default RT signal implementation reaches saturation at a slightly smaller load than signal-per-fd, and this is more pronounced for the 6000 idle connections. We will discuss this point in more detail below.

Figures 9(a), (b) and (c) plot the average response times of the various servers with increasing load for 252 and 6000 idle connections (1B and 6KB reply sizes) respectively. Figure 9(a) shows that `select()` reaches overload at a relatively low load, while the other mechanisms get overloaded at much higher loads. In figures 9(b) and (c), `select()` shows high response times for all loads and is thus overloaded for all the points in the graph. These plots complement figures 6(a), 7(a) and 8(a), which show the throughput for these cases. The figures further show that the `/dev/poll` server achieves small response times at low loads, but under overload, it offers much higher response times compared to the other mechanisms. Thus, its overload behavior is again seen to be very poor. The interesting point in figure 9(a) is that multi-accept `select` is able to provide a low response time upto very high loads. Figure 9(b) shows an even more interesting behavior of multi-accept `select` — its response time actually *decreases* with increasing load until it hits overload. This behavior clearly shows the load amortization occurring for multi-accept `select`, so that more useful work being extracted for the same `select()` call overhead translates to lower *average* response times. In figure 9(c), multi-accept `select` shows higher response times, but these increase only gradually with load, which again shows that the `select()` cost is being amortized. Finally, the two RT signal implementations have the lowest response times until they get overloaded, which is expected as they have the lowest overhead. Once again, these graphs show that the default RT signal based server reaches overload slightly earlier than the signal-per-fd server for the 1 byte reply size cases.

Next, we will try to understand these results, and in particular, we will focus on the behavior of multi-accept `select` and the two implementations of RT signals.

4.2 Discussion

From the results of our comparative study, we get the following insights into the behavior of the various mechanisms:

- Using multi-accept `select` increases the throughput of the `select()` based server substantially. This is because `select()` is basically a state notification mechanism. Thus, when it returns the listening socket as ready, it means there are new connections queued up on the listen queue. Using multiple `accept()` calls at this point drains the listen queue without having to call `select()` multiple times. This helps prevent the high cost of using multiple `select()` calls for identifying new connections. Once new connections are efficiently identified and added to the interest set, under high load, `select()` would have large number of active connections to report each time it is called. Thus, its cost would be amortized as the server could perform more useful work per `select()` call. This has to do with the fact that the ready set returned by `select()` would be denser and hence, the scanning cost of `select()` is utilized better. Also, the more useful work the server does on each call to `select()`, the less often it needs to be called. Hence, the server is able to identify more connections and extract more useful work, and thus achieves a higher throughput. Note that the overhead is still high — only the overhead is being *better utilized* now. This amortization is especially visible from figure 8 and figure 9(c), where the reply sizes are large, and hence, there is more work per connection.

The high throughput achieved by the multi-accept `select` server is contrary to conventional wisdom, according to which `select()` based servers should perform poorly under high loads in terms of their throughput as well. While this is true for a simple `select()` based server, our results show that implementing the server more carefully can help us achieve better performance.

- The behavior of the servers running on the default RT signal implementation and the signal-per-fd implementation are very similar until saturation. This is understandable as there are very few signal queue overflows under low loads, and hence, the two schemes work essentially the same. To verify that this is indeed the case, in table 1, we have tabulated the number of SIGIOs and the number of `sigwaitinfo()` calls for the default RT signal based server. These numbers correspond to some

Request Rate (req/s)	252 idle connections		6000 idle connections	
	No. of sigwaitinfos	No. of SIGIOS	No. of sigwaitinfos	No. of SIGIOS
2800	504728	0	504474	0
3000	540576	0	540792	0
3200	10538	1526	19	19
3400	40	40	16	16
3600	40	40	14	14
3800	39	39	13	13
4000	39	39	13	13

Table 1: Signal queue overflows under high loads

of the loads for 252 and 6000 idle connections respectively, with reply size of 1 byte. The number of `sigwaitinfo()`s denotes the number of times the server dequeued a signal from the signal queue, and the number of SIGIOS represents the number of signal queue overflows. As can be seen from the table, under low loads, there are no SIGIOS and hence, no signal queue overflows. On the other hand, at high loads, all the `sigwaitinfo()` calls result in SIGIOS. This indicates that the signal queue is overflowing all the time, and hence, the server has to fall back on an alternative mechanism to perform its network I/O under high loads. The fallback mechanism used in our server was multi-accept `select`². Hence, under high loads, the default RT signal server behavior is identical to that of the multi-accept `select` server, as was seen from the throughput and the response time plots.

- As noted earlier, for the 1 byte reply sizes, the signal-per-fd server reached overload at slightly higher load compared to the default RT signal server. In particular, the default RT signal based server saturated at about 3200 req/s, which corresponds to the high number of `sigwaitinfo()`s resulting in SIGIOS at this load, as can be seen from table 1. Thus, preventing signal queue overflows seems to make the server sustain slightly higher loads before getting saturated. Once the signal-per-fd server becomes saturated, its throughput is bounded by the amount of useful work it can amortize over each signal notification, even though it does not suffer from signal queue overflows. Recall that similar to `select()`, signal-per-fd is also a state-notification mechanism — hence the server can extract more work per signal compared to the default event-notification mechanism. Thus, its throughput is comparable to that of

multi-accept `select` under overload, even though its peak throughput is slightly smaller, as `sigwaitinfo()` still returns only one signal per call.

To summarize, we find that RT signals are an efficient mechanism in terms of overhead, and under saturation, their throughput is determined by the fallback mechanism being used to handle signal queue overflows. We find that `select()` system call can give high throughput if we use multiple `accept()`s to identify more new connections per `select()` call. This is in spite of the fact that `select()` based servers have high CPU overhead. Finally, signal-per-fd has a behavior almost identical to that of the default RT signal implementation in terms of overhead and throughput, but it is able to sustain slightly higher load before becoming overloaded. Further, it helps reduce the complexity of the server to a large extent. This is because we do not have to worry about using alternative event-dispatch mechanisms, and state maintenance also becomes much easier.

5 Conclusion

In this paper, we first discussed some of the common event-dispatch mechanisms employed by Internet servers. We focussed on the mechanisms available in the Linux kernel, and measured their performance in terms of the overhead and throughput of a minimal Web server. Our comparative studies showed that RT signals are a highly efficient mechanism in terms of their dispatch overhead and also provide good throughput compared to mechanisms like `select()` and `/dev/poll`. In particular, the overhead of RT signals is independent of the number of connections being handled by the server, and depends only on the active I/O being performed by it. But, an RT signal based server can suffer from signal queue overflows. Handling such overflows leads to complexity in the server implementation and also potential performance penalties under high loads. To overcome these drawbacks, we proposed a scheme called *signal-per-fd*, which is an enhancement to the default RT signal

²We cannot simply use `select()` with single `accept()` in this situation because, to prevent any potential deadlocks, we have to ensure that no event is lost, and hence, we need to clear the listen queue completely.

implementation in the Linux kernel. This enhancement was shown to significantly reduce the complexity of a server implementation, increasing its robustness under high load, and also potentially increasing its throughput. Overall, we conclude that RT signals are a highly scalable event-dispatch mechanism and servers based on these signals can also be substantially simplified when coupled with the signal-per-fd enhancement.

Another interesting result of our study was the performance of `select()` based servers under high loads. According to conventional wisdom, `select()` based servers have high overhead and thus, perform very poorly under high loads in terms of the server throughput as well. Our experiments with the *multi-accept* variant of a `select()` based server show that though `select()` does have high dispatch overhead, this overhead can be amortized better by performing more useful work per `select()` call, resulting in a high throughput even under heavy load conditions. Thus, we conclude that even a `select()` based server can be made to scale substantially if its overhead is better utilized to perform more useful work.

Acknowledgements

We would like to thank Martin Arlitt for providing us with large number of client machines and helping us set up the test-bed for our experiments. We would also like to thank the anonymous reviewers and our shepherd Greg Ganger whose invaluable comments and suggestions helped us improve this paper immensely.

References

- [1] Gaurav Banga and Jeffrey C. Mogul. Scalable kernel performance for Internet servers under realistic loads. In *Proceedings of the USENIX Annual Technical Conference*, June 1998.
- [2] Gaurav Banga, Jeffrey C. Mogul, and Peter Druschel. A scalable and explicit event delivery mechanism for UNIX. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.
- [3] Gordon Bell and Jim Gemmell. On-ramp prospects for the Information Superhighway Dream. *Communications of the ACM*, 39(7):55–61, July 1996.
- [4] Z. Brown. phhttpd. <http://www.zabbo.net/phhttpd>, November 1999.
- [5] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2068, January 1997.
- [6] Bill O. Gallmeister. *POSIX.4: Programming for the Real World*. O'Reilly, 1995.
- [7] James C. Hu, Irfan Pyarali, and Douglas C. Schmidt. Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-speed Networks. In *Proceedings of the Second IEEE Global Internet Conference*, November 1997.
- [8] David Mosberger and Tai Jin. httpf – A Tool for Measuring Web Server Performance. In *Proceedings of the SIGMETRICS Workshop on Internet Server Performance*, June 1998.
- [9] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.
- [10] Niels Provos and Chuck Lever. Scalable Network I/O in Linux. In *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*, June 2000.
- [11] Niels Provos, Chuck Lever, and Stephen Tweedie. Analyzing the Overload Behavior of a Simple Web Server. In *Proceedings of the Fourth Annual Linux Showcase and Conference*, October 2000.
- [12] Jeffrey Richter. *Advanced Windows*. Microsoft Press, third edition, 1997.
- [13] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.
- [14] Solaris 8 man pages for `poll(7d)`. <http://docs.sun.com:80/ab2/coll.40.6/REFMAN7/ @Ab2PageView/55123?Ab2Lang=C&Ab2Enc=iso-8859-1>.
- [15] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1990.
- [16] thttpd – tiny/turbo/throttling HTTP server. <http://www.acme.com/software/thttpd>.
- [17] Answers from Planet TUX: Ingo Molnar Responds (*interview*). <http://slashdot.org/interviews/00/07/20/1440204.shtml>.
- [18] Zeus Web Server. <http://www.zeustech.net/products/ws>.

Virtual-Time Round-Robin: An O(1) Proportional Share Scheduler

Jason Nieh Chris Vaill Hua Zhong

*Department of Computer Science
Columbia University*

{nieh, cvaill, huaz}@cs.columbia.edu

Abstract

Proportional share resource management provides a flexible and useful abstraction for multiplexing time-shared resources. However, previous proportional share mechanisms have either weak proportional sharing accuracy or high scheduling overhead. We present Virtual-Time Round-Robin (VTRR), a proportional share scheduler that can provide good proportional sharing accuracy with $O(1)$ scheduling overhead. VTRR achieves this by combining the benefits of fair queueing algorithms with a round-robin scheduling mechanism. Unlike many other schedulers, VTRR is simple to implement. We have implemented a VTRR CPU scheduler in Linux in less than 100 lines of code. Our performance results demonstrate that VTRR provides accurate proportional share allocation with constant, sub-microsecond scheduling overhead. The scheduling overhead using VTRR is two orders of magnitude less than the standard Linux scheduler for large numbers of clients.

1 Introduction

Proportional share resource management provides a flexible and useful abstraction for multiplexing scarce resources among users and applications. The basic idea is that each client has an associated weight, and resources are allocated to the clients in proportion to their respective weights. Because of its usefulness, many proportional share scheduling mechanisms have been developed [3, 8, 10, 14, 16, 18, 25, 28, 30, 33, 34]. In addition, higher-level abstractions have been developed on top of these proportional share mechanisms to support flexible, modular resource management policies [30, 33].

Proportional share scheduling mechanisms were first developed decades ago with the introduction of weighted round-robin scheduling [29]. Later, fair-share algorithms based on controlling priority values were developed and incorporated into some UNIX operating systems [10, 16, 18]. These earlier mechanisms were typi-

cally fast, requiring only constant time to select a client for execution. However, they were limited in the accuracy with which they could achieve proportional sharing. As a result starting in the late 1980s, fair queueing algorithms were developed [3, 8, 14, 25, 28, 30, 33, 34], first for network packet scheduling and later for CPU scheduling. These algorithms provided better proportional sharing accuracy. However, the time to select a client for execution using these algorithms grows with the number of clients. Most implementations require linear time to select a client for execution. For server systems which may service large numbers of clients, the scheduling overhead of linear time algorithms can waste more than 20 percent of system resources [5] for large numbers of clients. Hierarchical data structures can be used to reduce the selection time complexity, but they are not generally used as they are often less efficient in practice. This is because they add implementation complexity and their performance depends on being able to balance the data structures efficiently.

In this paper, we introduce VTRR, a Virtual-Time Round-Robin scheduler for proportional share resource management. VTRR combines the benefits of low overhead round-robin execution with high accuracy virtual-time allocations. It provides accurate control over client computation rates, and it can schedule clients for execution in $O(1)$ time. The constant scheduling overhead makes VTRR particularly suitable for server systems that must manage large numbers of clients. VTRR is simple to implement and can be easily incorporated into existing scheduling frameworks in commercial operating systems. We have implemented a prototype VTRR CPU scheduler in Linux in less than 100 lines of code. We have compared our VTRR Linux prototype against schedulers commonly used in practice and research, including the standard Linux scheduler [2] and fair queueing. Our performance results on micro-benchmarks and real applications demonstrate that VTRR delivers excellent proportional share control with lower scheduling overhead than other approaches.

This paper is organized as follows: Section 2 dis-

cusses background and related work. Section 3 presents the VTRR scheduling algorithm. Section 4 describes our prototype Linux implementation. Section 5 presents performance results from both simulation studies and real kernel measurements that compare VTRR against weighted round-robin, fair queueing, and standard Linux scheduling. Finally, we present some concluding remarks and directions for future work.

2 Background

Previous proportional sharing mechanisms can be classified into four categories: those that are fast but have weaker proportional fairness guarantees, those that map well to existing scheduler frameworks in current commercial operating systems but have no well-defined proportional fairness guarantees, those that have strong proportional fairness guarantees and higher scheduling overhead, and those that have weaker proportional fairness guarantees but have higher scheduling overhead. The four categories correspond to round-robin, fair-share, fair queueing, and lottery mechanisms.

To discuss these different approaches, we first present in Section 2.1 a simple proportional share model for scheduling a time-multiplexed resource and more precisely define the notion of proportional fairness. In Sections 2.2 to 2.4, we use this background to explain the round-robin, fair-share, fair queueing, and lottery sharing mechanisms in further detail. We briefly mention other related work in Section 2.6.

2.1 Proportional Fairness

Proportional share scheduling has a clear colloquial meaning: given a set of clients with associated weights, a proportional share scheduler should allocate resources to each client in proportion to its respective weight. In this paper, we use the term share and weight interchangeably. Without loss of generality, we can model the process of scheduling a time-multiplexed resource among a set of clients in two steps: 1) the scheduler orders the clients in a queue, 2) the scheduler runs the first client in the queue for its *time quantum*, which is the maximum time interval the client is allowed to run before another scheduling decision is made. Note that the time quantum is typically expressed in time units of constant size determined by the hardware. As a result, we refer to the units of time quanta as time units (tu) in this paper rather than an absolute time measure such as seconds.

Based on the above scheduler model, a scheduler can achieve proportional sharing in one of two ways. One way is to adjust the frequency that a client is selected to

run by adjusting the position of the client in the queue so that it ends up at the front of the queue more or less often. The other way is to adjust the size of the time quantum of a client so that it runs longer for a given allocation. The manner in which a scheduler determines how often a client runs and how long a client runs directly affects the accuracy and scheduling overhead of the scheduler.

A proportional share scheduler is more accurate if it allocates resources in a manner that is more proportionally fair. We can formalize this notion of proportional fairness in more technical terms. The definition we use is a simple one that suffices for our discussion; more extended definitions are presented in [12, 15, 26, 32]. Our definition draws heavily from the ideal sharing mechanism GPS [19]. To simplify the discussion, we assume that clients do not sleep or block and can consume whatever resources they are allocated.

We first define *perfect fairness*, an ideal state in which each client has received service exactly proportional to its share. We denote the proportional share of client A as S_A , and the amount of service received by client A during the time interval (t_1, t_2) as $W_A(t_1, t_2)$. Formally, a proportional sharing algorithm achieves perfect fairness for time interval (t_1, t_2) if, for any client A ,

$$W_A(t_1, t_2) = (t_2 - t_1) \frac{S_A}{\sum_i S_i} \quad (1)$$

If we had an ideal system in which all clients could consume their resource allocations simultaneously, then an ideal proportional share scheduler could maintain the above relationship for all time intervals. However, in scheduling a time-multiplexed resource in time units of finite size, it is not possible for a scheduler to be perfectly proportionally fair as defined by Equation 1 for all intervals.

Although no real-world scheduling algorithm can maintain perfect fairness, some algorithms stay closer to perfect fairness than others. To evaluate the fairness performance of a proportional sharing mechanism, we must quantify how close an algorithm gets to perfect fairness. We can use a variation of Equation 1 to define the *service time error* $E_A(t_1, t_2)$ for client A over interval (t_1, t_2) . The error is the difference between the amount time allocated to the client during interval (t_1, t_2) under the given algorithm, and the amount of time that would have been allocated under an ideal scheme that maintains perfect fairness for all clients over all intervals. Service time error is computed as:

$$E_A(t_1, t_2) = W_A(t_1, t_2) - (t_2 - t_1) \frac{S_A}{\sum_i S_i} \quad (2)$$

A positive service time error indicates that a client has received more than its ideal share over an interval; a negative error indicates that a client has received less. To be

precise, the error E_A measures how much time client A has received beyond its ideal allocation.

The goal of a proportional share scheduler should be to minimize the allocation error between clients. In this context, we now consider how effectively different classes of proportional share algorithms are in minimizing this allocation error.

2.2 Round-Robin

One of the oldest, simplest and most widely used proportional share scheduling algorithms is round-robin. Clients are placed in a queue and allowed to execute in turn. When all client shares are equal, each client is assigned the same size time quantum. In the weighted round-robin case, each client is assigned a time quantum equal to its share. A client with a larger share, then, effectively gets a larger quantum than a client with a small share. Weighted round-robin (WRR) provides proportional sharing by running all clients with the same frequency but adjusting the size of their time quanta. A more recent variant called deficit round-robin [28] has been developed for network packet scheduling with similar behavior to a weighted round-robin CPU scheduler.

WRR is simple to implement and schedules clients in $O(1)$ time. However, it has a relatively weak proportional fairness guarantee as its service ratio error can be quite large. Consider an example in which 3 clients A, B, and C, have shares 3, 2, and 1, respectively. WRR will execute these clients in the following order of time units: A, A, A, B, B, C. The error in this example gets as low as -1 tu and as high as $+1.5$ tu. The real trouble comes with large share values: if the shares in the previous example are changed to 3000, 2000, and 1000, the error ranges instead from -1000 to $+1500$ tu. A large error range like this illustrates the major drawback of round-robin scheduling: each client gets all service due to it all at once, while other clients get no service. After a client has received all its service, it is well ahead of its ideal allocation (it has a high positive error), and all other clients are behind their allocations (they have low negative errors).

2.3 Fair-Share

Fair-share schedulers [10, 16, 18] arose as a result of a need to provide proportional sharing among users in a way compatible with a UNIX-style time-sharing framework. In UNIX time-sharing, scheduling is done based on multi-level feedback with a set of priority queues. Each client has a priority which is adjusted as it executes. The scheduler executes the client with the highest priority. The idea of fair-share was to provide proportional sharing among users by adjusting the priorities of a

user's clients in a suitable way. Fair-share provides proportional sharing by effectively running clients at different frequencies, as opposed to WRR which only adjusts the size of the clients' time quanta. Fair-share schedulers were compatible with UNIX scheduling frameworks and relatively easy to deploy in existing UNIX environments. Unlike round-robin scheduling, the focus was on providing proportional sharing to groups of users as opposed to individual clients. However, the approaches were often ad-hoc and it is difficult to formalize the proportional fairness guarantees they provided. Empirical measurements show that these approaches only provide reasonable proportional fairness over relatively large time intervals [10]. It is almost certainly the case that the allocation errors in these approaches can be very large.

The priority adjustments done by fair-share schedulers can generally be computed quickly in $O(1)$ time. In some cases, the schedulers need to do an expensive periodic re-adjustment of all client priorities, which required $O(N)$ time, where N is the number of clients.

2.4 Fair Queueing

Fair queueing was first proposed by Demers et. al. for network packet scheduling as Weighted Fair Queueing (WFQ) [8], with a more extensive analysis provided by Parekh and Gallager [25], and later applied by Waldspurger and Weihs to CPU scheduling as stride scheduling [33]. WFQ introduced the idea of a virtual finishing time (VFT) to do proportional sharing scheduling. To explain what a VFT is, we first explain the notion of virtual time. The *virtual time* of a client is a measure of the degree to which a client has received its proportional allocation relative to other clients. When a client executes, its virtual time advances at a rate inversely proportional to the client's share. In other words, the virtual time of a client A at time t is the ratio of $W_A(t)$ to S_A :

$$VT_A(t) = \frac{W_A(t)}{S_A} \quad (3)$$

Given a client's virtual time, the client's *virtual finishing time* (VFT) is defined as the virtual time the client would have after executing for one time quantum. WFQ then schedules clients by selecting the client with the smallest VFT. This is implemented by keeping an ordered queue of clients sorted from smallest to largest VFT, and then selecting the first client in the queue. After a client executes, its VFT is updated and the client is inserted back into the queue. Its position in the queue is determined by its updated VFT. Fair queueing provides proportional sharing by running clients at different frequencies by adjusting the position in at which each client is inserted back into the queue; the same size time

quantum is used for all clients.

To illustrate how this works, consider again the example in which 3 clients A, B, and C, have shares 3, 2, and 1, respectively. Their initial VFTs are then 1/3, 1/2, and 1, respectively. WFQ would then execute the clients in the following order of time units: A, B, A, B, C, A. In contrast to WRR, WFQ's service time error ranges from $-5/6$ to $+1$ tu in this example, which is less than the allocation error of -1 to $+1.5$ tu for WRR. The difference between WFQ and WRR is greatly exaggerated if larger share values are chosen: if we make the shares 3000, 2000, and 1000 instead of 3, 2, and 1, WFQ has the same service time error range while WRR's error range balloons to -1000 to $+1500$ tu.

It has been shown that WFQ guarantees that the service time error for any client never falls below -1 , which means that a client can never fall behind its ideal allocation by more than a single time quantum [25]. More recent fair queueing algorithms [3, 30] provide more accurate proportional sharing (by also guaranteeing an upper bound on error) at the expense of additional scheduling overhead. Fair queueing provides stronger proportional fairness guarantees than round-robin or fair-share scheduling. Unfortunately, fair queueing is more difficult to implement, and the time it takes to select a client to execute is $O(N)$ time for most implementations, where N is the number of clients. With more complex data structures, it is possible to implement fair queueing such that selection of a client requires $O(\log N)$ time. However, the added difficulty of managing complex data structures in kernel space causes most implementers of fair queueing to choose the more straightforward $O(N)$ implementation.

2.5 Lottery

Lottery scheduling was proposed by Waldspurger and Weihs [33] after WFQ was first developed. In lottery scheduling, each client is given a number of tickets proportional to its share. A ticket is then randomly selected by the scheduler and the client that owns the selected ticket is scheduled to run for a time quantum. Like fair queueing, lottery scheduling provides proportional sharing by running clients at different frequencies by adjusting the position in at which each client is inserted back into the queue; the same size time quantum is typically used for all clients.

Lottery scheduling is somewhat simpler to implement than fair queueing, but has the same high scheduling overhead as fair queueing, $O(N)$ for most implementations or $O(\log N)$ for more complex data structures. However, because lottery scheduling relies on the law of large numbers for providing proportional fairness, its accuracy is much worse than WFQ [33], and is also worse

than WRR for smaller share values.

2.6 Other Related Work

Higher-level resource management abstractions have also been developed [1, 33], and a number of these abstractions can be used with proportional share scheduling mechanisms. This work is complementary to our focus here on the underlying scheduling mechanisms. Other scheduling work has also been done in supporting clients with real-time requirements [4, 6, 13, 17, 20, 21, 22, 23, 24] and improving the response time of interactive clients [9, 11, 24]. Considering these issues in depth is beyond the scope of this paper.

3 VTRR Scheduling

VTRR is an accurate, low-overhead proportional share scheduler for multiplexing time-shared resources among a set of clients. VTRR combines the benefit of low overhead round-robin scheduling with the high accuracy mechanisms of virtual time and virtual finishing time used in fair queueing algorithms. At a high-level, the VTRR scheduling algorithm can be briefly described in three steps:

1. Order the clients in the run queue from largest to smallest share. Unlike fair queueing, a client's position on the run queue only changes when its share changes, an infrequent event, not on each scheduling decision.
2. Starting from the beginning of the run queue, run each client for one time quantum in a round-robin manner. VTRR uses the fixed ordering property of round-robin in order to choose in constant time which client to run. Unlike round-robin, the time quantum is the same size for all clients.
3. In step 2, if a client has received more than its proportional allocation, skip the remaining clients in the run queue and start running clients from the beginning of the run queue again. Since the clients with larger share values are placed first in the queue, this allows them to get more service than the lower-share clients at the end of the queue.

To provide a more in depth description of VTRR, we first define the state VTRR associates with each client, then describe precisely how VTRR uses that state to schedule clients. In VTRR, a client has five values associated with its execution state: share, virtual finishing time, time counter, id number, and run state. A client's *share* defines its resource rights. Each client receives

a resource allocation that is directly proportional to its share. A client's *virtual finishing time* (VFT) is defined in the same way as in Section 2.4. Since a client has a VFT, it also has an implicit virtual time. A client's VFT advances at a rate proportional to its resource consumption divided by its share. The VFT is used to decide when VTRR should reset to the first client in the queue. This is described in greater detail in Section 3.1, below. A client's *time counter* ensures that the pattern of allocations is periodic, and that perfect fairness is achieved at the end of each period. Specifically, the time counter tracks the number of quanta the client must receive before the period is over and perfect fairness is reached. A client's *id number* is a unique client identifier that is assigned when the client is created. A client's *run state* is an indication of whether or not the client can be executed. A client is *Runnable* if it can be executed, and not runnable if it cannot. For example for a CPU scheduler, a client would not be runnable if it is blocked waiting for I/O and cannot execute.

3.1 Basic VTRR Algorithm

We will initially only consider runnable clients in our discussion of the basic VTRR scheduling algorithm. We will discuss dynamic changes in a client's run state in Section 3.2. VTRR maintains the following scheduler state: time quantum, run queue, total shares, and queue virtual time. As discussed in Section 2.1, the *time quantum* is the duration of a standard time slice assigned to a client to execute. The *run queue* is a sorted queue of all runnable clients ordered from largest to smallest share client. Ties can be broken either arbitrarily or using the client id numbers, which are unique. The *total shares* is the sum of the shares of all runnable clients. The *queue virtual time* (QVT) is a measure of what a client's VFT should be if it has received exactly its proportional share allocation.

Previous work in the domain of packet scheduling provides the theoretical basis for the QVT [8, 25]. The QVT advances whenever a client executes at a rate inversely proportional to the total shares. If we denote the system time quantum as Q and the share of client i as S_i , then the QVT is updated as follows:

$$QVT(t+Q) = QVT(t) + \frac{Q}{\sum_i S_i} \quad (4)$$

The difference between the QVT and a client's virtual time is a measure of whether the respective client has consumed its proportional allocation of resources. If a client's virtual time is equal to the queue virtual time, it is considered to have received its proportional allocation of resources. An earlier virtual time indicates that the client has used less than its proportional share. Sim-

ilarly, a later virtual time indicates that it has used more than its proportional share. Since the QVT advances at the same rate for all clients on the run queue, the relative magnitudes of the virtual times provide a relative measure of the degree to which each client has received its proportional share of resources.

First, we explain the role of the time counters in VTRR. In relation to this, we define a *scheduling cycle* as a sequence of allocations whose length is equal to the sum of all client shares. For example, for a queue of three clients with shares 3, 2, and 1, a scheduling cycle is a sequence of 6 allocations. The time counter for each client is reset at the beginning of each scheduling cycle to the client's share value, and is decremented every time a client receives a time quantum. VTRR uses the time counters to ensure that perfect fairness is attained at the end of every scheduling cycle. At the end of the cycle, every counter is zero, meaning that for each client A , the number of quanta received during the cycle is exactly S_A , the client's share value. Clearly, then, each client has received service proportional to its share. In order to guarantee that all counters are zero at the end of the cycle, we enforce an invariant on the queue, called the *time counter invariant*: we require that, for any two consecutive clients in the queue A and B , the counter value for B must always be no greater than the counter value for A .

The VTRR scheduling algorithm starts at the beginning of the run queue and executes the first client for one time quantum. We refer to the client selected for execution as the *current client*. Once the current client has completed its time quantum, its time counter is decremented by one and its VFT is incremented by the time quantum divided by its share. If we denote the system time quantum as Q , the current client's share as S_C , and the current client's VFT as $VFT_C(t)$, $VFT_C(t)$ is updated as follows:

$$VFT_C(t+Q) = VFT_C(t) + \frac{Q}{S_C} \quad (5)$$

The scheduler then moves on to the next client in the run queue. First, the scheduler checks for violation of the time counter invariant: if the counter value of the next client is greater than the counter of the current client, the scheduler makes the next client the current client and executes it for a quantum, without question. This causes its counter to be decremented, preserving the invariant. If the next client's counter is not greater than the current client's counter, the time counter invariant cannot be violated whether the next client is run or not, so the scheduler makes a decision using virtual time: the scheduler compares the VFT of the next client with the QVT the system would have after the next time quantum a client executes. We call this comparison the *VFT*

inequality. If we denote the system time quantum as Q , the current client's VFT as $VFT_C(t)$, and its share as S_C , the VFT inequality is true if:

$$VFT_C(t) - QVT(t+Q) < \frac{Q}{S_C} \quad (6)$$

If the VFT inequality is true, the scheduler selects and executes the next client in the run queue for one time quantum and the process repeats with the subsequent clients in the run queue. If the scheduler reaches a point in the run queue when the VFT inequality is not true, the scheduler returns to the beginning of the run queue and selects the first client to execute. At the end of the scheduling cycle, when the time counters of all clients reach zero, the time counters are all reset to their initial values corresponding to the respective client's share, and the scheduler starts from the beginning of the run queue again to select a client to execute. Note that throughout this scheduling process, the ordering of clients on the run queue does not change.

To illustrate how this works, consider again the example in which 3 clients A, B, and C, have shares 3, 2, and 1, respectively. Their initial VFTs are then 1/3, 1/2, and 1, respectively. VTRR would then execute the clients in the following repeating order of time units: A, B, C, A, B, A. In contrast to WRR and WFQ, VTRR has a maximum allocation error between A and B of 1/3 tu in this example. This allocation error is much better than WRR and comparable to WFQ.

Since VTRR simply selects each client in turn to execute, selecting a client for execution can be done in $O(1)$ time. We defer a more detailed discussion of the complexity of VTRR to Section 3.3.

3.2 VTRR Dynamic Considerations

In the previous section, we presented the basic VTRR scheduling algorithm, but we did not discuss how VTRR deals with dynamic considerations that are a necessary part of any on-line scheduling algorithm. We now discuss how VTRR allows clients to be dynamically created, terminated, change run state, and change their share assignments.

We distinguish between clients that are runnable and not runnable. As mentioned earlier, clients that are runnable can be selected for execution by the scheduler, while clients that are not runnable cannot. Only runnable clients are placed in the run queue. With no loss of generality, we assume that a client is created before it can become runnable, and a client becomes not runnable before it is terminated. As a result, client creation and termination have no affect on the VTRR run queue.

When a client becomes runnable, it is inserted into the run queue so that the run queue remains sorted from

largest to smallest share client. Ties can be broken either arbitrarily or using the unique client id numbers. One issue remains, which is how to determine the new client's initial VFT. When a client is created and becomes runnable, it has not yet consumed any resources, so it is neither below or above its proportional share in terms of resource consumption. As a result, we set the client's implicit virtual time to be the same as the QVT. We can then calculate the VFT of a new client A with share S_A as:

$$VFT_A(t) = QVT_A(t) + \frac{Q}{S_A} \quad (7)$$

After a client is executed, it may become not runnable. If the client is the current client and becomes not runnable, it is preempted and another client is selected by the scheduler using the basic algorithm described in Section 3.1. The client that is not runnable is removed from the run queue. If the client becomes not runnable and is not the current client, the client is simply removed from the run queue. While the client is not runnable, its VFT is not updated. When the client is removed from the run queue, it records the client that was before it on the run queue, and the client that was after it on the run queue. We refer to these clients as the *last-previous* client and *last-next* client, respectively.

When a client that is not runnable becomes runnable again, VTRR inserts the now runnable client back into the run queue. If the client's references to its last-previous and last-next client are still valid, it can use those references to determine its position in the run queue. If either the last-previous or the last-next reference is not valid, VTRR then simply traverses the run queue to find the insertion point for the now runnable client.

Determining whether the last-previous and last-next references are valid can be done efficiently as follows. The last-previous and last-next client references are valid if both clients have not exited and are runnable, if there are no clients between them on the run queue, and if the share of the newly-runnable client is no more than the last-previous client and no less than the last-next client. Care must be taken, however, to ensure that the last-previous and last-next references are still valid before dereferencing them: if either client has exited and been deallocated, last-previous and last-next may no longer refer to valid memory regions. To deal with this, a hash table can be kept that stores identifiers of valid clients. Hash function collisions can be resolved by simple replacement, so the table can be implemented as an array of identifiers. A client's identifier is put into the table when it is created, and deleted when the client exits. The last-previous and last-next pointers are not dereferenced, then, unless the identifier of the last-previous and

last-next clients exist in the hash table. As described in Section 4, the use of a hash table was not necessary in our Linux VTRR implementation.

Once the now runnable client has been inserted in the run queue, the client's VFT must be updated. The update is analogous to the VFT initialization used when a new client becomes runnable. The difference is that we also account for the client's original VFT in updating the VFT. If we denote the original VFT of a client A as $VFT_A(t')$, then the client's VFT is updated as follows:

$$VFT_A(t) = \text{MAX}\{QVFT_A(t) + \frac{Q}{S_A}, VFT_A(t')\} \quad (8)$$

This treats a client that has been not runnable for a while like a new client that has not yet executed. At the same time, the system keeps track of the client's VFT so that if it that has recently used more than its proportional allocation, it cannot somehow game the system by making itself not runnable and becoming runnable again.

We use an analogous policy to set the initial value of a client's time counter. A client's time counter tracks the number of quanta due to the client before the end of the current scheduling cycle, and is reset at the beginning of each new cycle. We set the time counter of a newly-inserted client to a value which will give it the correct proportion of remaining quanta in this cycle. The counter C_A for the new client A is computed:

$$C_A = \frac{S_A}{\sum_i S_i} \sum_i C_i \quad (9)$$

Note that this is computed *before* client A is inserted, so S_A is not included in the $\sum_i S_i$ summation.

This value is modified by a rule similar to the rule enacted for the VFT: we require that a client cannot come back in the same cycle and receive a larger time count than it had previously. Therefore, if the client is being inserted during the same cycle in which it was removed, the counter is set to the minimum of C_A and the previous counter value. Finally, to preserve the time counter invariant (as described in Section 3.1), the counter value must be restricted to be between the time counter values of the clients before and after the inserted client.

If a client's share changes, there are two cases to consider based on the run state of the client. If the client is not runnable, no run queue modifications are needed. If the client is runnable and its share changes, the client's position in the run queue may need to be changed. This operation can be simplified by removing the client from the run queue, changing the share, and then reinserting it. Removal and insertion can then be performed just as described above.

3.3 Complexity

The primary function of a scheduler is to select a client to execute when the resource is available. A key benefit of VTRR is that it can select a client to execute in $O(1)$ time. To do this, VTRR simply has to maintain a sorted run queue of clients and keep track of its current position in the run queue. Updating the current run queue position and updating a client's VFT are both $O(1)$ time operations. While the run queue needs to be sorted by client shares, the ordering of clients on the run queue does not change in the normal process of selecting clients to execute. This is an important advantage over fair queueing algorithms, in which a client needs to be reinserted into a sorted run queue after each time it executes. As a result, fair queueing has much higher complexity than VTRR, requiring $O(N)$ time to select a client to execute, or $O(\log N)$ time if more complex data structures are used (but this is rarely implemented in practice).

When all clients on the run queue have zero counter values, VTRR resets the counter values of all clients on the run queue. The complete counter reset takes $O(N)$ time, where N is the number of clients. However, this reset is done at most once every N times the scheduler selects a client to execute (and much less frequently in practice). As a result, the reset of the time counters is amortized over many client selections so that the effective running time of VTRR is still $O(1)$ time. In addition, the counter resets can be done incrementally on the first pass through the run queue with the new counter values.

In addition to selecting a client to execute, a scheduler must also allow clients to be dynamically created and terminated, change run state, and change scheduling parameters such as a client's share. These scheduling operations typically occur much less frequently than client selection. In VTRR, operations such as client creation and termination can be done in $O(1)$ time since they do not directly affect the run queue. Changing a client's run state from runnable to not runnable can also be done in $O(1)$ time for any reasonable run queue implementation since all it involves is removing the respective client from the run queue. The scheduling operations with the highest complexity are those that involve changing a client's share assignment and changing a client's run state to runnable. In particular, a client typically becomes runnable after it is created or after an I/O operation that it was waiting for completes. If a client's share changes, the client's position in the run queue may have change as well. If a client becomes runnable, the client will have to be inserted into the run queue in the proper position based on its share. Using a doubly linked list run queue implementation, insertion into the sorted queue can require $O(N)$ time, where N is the number of

runnable clients. A priority queue implementation could be used for the run queue to reduce the insertion cost to $O(\log N)$, but probably does not have better overall performance than a simple sorted list in practice.

Because queue insertion is required much less frequently than client selection in practice, the queue insertion cost is not likely to dominate the scheduling cost. In particular, if only a constant number of queue insertions are required for every N times a client selection is done, then the effective cost of the queue insertions is still only $O(1)$ time. Furthermore, the most common scheduling operation that would require queue insertion is when a client becomes runnable again after it was blocked waiting on a resource. In this case, the insertion overhead can be $O(1)$ time if the last-previous client and last-next client references remain valid at queue insertion time. If the references are valid, then the position of the client is already known on the run queue so the scheduler does not have to find the insertion point.

An alternative implementation can be done that allows all queue insertions to be done in $O(1)$ time, if the range of share values is fixed in advance. The idea is similar to priority schedulers which have a fixed range of priority values and have separate run queue for each priority. Instead of using priorities, we can have a separate run queue for each share value and keep track of the run queues using an array. We can then find the queue corresponding to a client's share and insert the client at the end of the corresponding queue in $O(1)$ time. Such an implementation maps well to scheduling frameworks in a number of commercial operating systems, including Solaris [31] and Windows NT [7].

4 Implementation

We have implemented a prototype VTRR CPU scheduler in the Linux operating system. For this work, we used the Red Hat Linux version 6.1 distribution and the Linux version 2.2.12-20 kernel. We had to add or modify less than 100 lines of kernel code to complete the VTRR scheduler implementation. We describe our Linux VTRR implementation in further detail to illustrate how easy VTRR is to implement. These scheduling frameworks are commonly found in commercial operating systems. While VTRR can be used in a multiprocessor scheduling context, we only discuss the single CPU implementation here.

The Linux scheduling framework for a single CPU is based on a run queue implemented as a single doubly linked list. We first describe how the standard Linux scheduler works, and then discusses the changes we made to implement VTRR in Linux.

The standard Linux scheduler multiplexes a set of

clients that can be assigned different priorities. The priorities are used to compute a per client measure called *goodness* to schedule the set of clients. Each time the scheduler is called, the goodness value for each client in the run queue is calculated. The client with the highest goodness value is then selected as the next client to execute. In the case of ties, the first client with the highest goodness value is selected. Because the goodness of each client is calculated each time the scheduler is called, the scheduling overhead of the Linux scheduler is $O(N)$, where N is the number of runnable clients.

The standard way Linux calculates the goodness for all clients is based on a client's priority and counter. The counter is not the same as the time counter value used by VTRR, but is instead a measure of the remaining time left in a client's time quantum. The standard time unit used in Linux for the counter and time quantum is called a jiffy, which is 10 ms by default. The basic idea is that the goodness of a client is its priority plus its counter value. The client's counter is initially set equal to the client's priority, which has a value of 20 by default. Each time a client is executed for a jiffy, the client's counter is decremented. A client's counter is decremented until it drops below zero, at which point the client cannot be selected to execute. As a result, the default time quantum for each client is 21 jiffies, or 210 ms. When the counters of all runnable clients drop below zero, the scheduler resets all the counters to their initial value. There is some additional logic to support static priority real-time clients and clients that become not runnable, but an overview of the basic way in which the Linux scheduler works is sufficient for our discussion here. Further details are available elsewhere [2].

To implement VTRR in Linux, we reused much of the existing scheduling infrastructure. We used the same doubly linked list run queue structure as the standard Linux scheduler. The primary change to the run queue was sorting the clients from largest to smallest share. Rather than scanning all the clients when a scheduling decision needs to be made, our VTRR Linux implementation simply picks the next client in the run queue based on the VTRR scheduling algorithm.

One idiosyncrasy of the Linux scheduler that is relevant to this work is that the smallest counter value that may be assigned to a client is 1. This means that the smallest time quantum a client can have is 2 jiffies. To provide a comparable implementation of VTRR, the default time quantum used in our VTRR implementation is also 2 jiffies, or 20 ms.

In addition to the VTRR client state, two fields that were added to the standard client data structure in Linux were last-previous and last-next pointers which were used to optimize run queue insertion efficiency. In the Linux 2.2 kernel, memory for the client data structures

is statically allocated, and never reclaimed for anything other than new client data structures. Therefore, in our implementation, we were free to reference the last-next and last-previous pointers to check their validity, as they always refer to some client's data; the hash table method described in Section 3.2 was unnecessary.

5 Measurements and Results

To demonstrate the effectiveness of VTRR, we have quantitatively measured and compared its performance against other leading approaches from both industrial practice and research. We have conducted both extensive simulation studies and detailed measurements of real kernel scheduler performance on real applications.

We conducted simulation studies to compare the proportional sharing accuracy of VTRR against both WRR and WFQ. We used a simulator for these studies for two reasons. First, our simulator enabled us to isolate impact of the scheduling algorithms themselves and purposefully do not include the effects of other activity present in an actual kernel implementation. Second, our simulator enabled us to examine the scheduling behavior of these different algorithms across hundreds of thousands of different combinations of clients with different share values. It would have been much more difficult to obtain this volume of data in a repeatable fashion from just measurements of a kernel scheduler implementation. Our simulation results are presented in Section 5.1.

We also conducted detailed measurements of real kernel scheduler performance by comparing our prototype VTRR Linux implementation against both the standard Linux scheduler and a WFQ scheduler. In particular, comparing against the standard Linux scheduler and measuring its performance is important because of its growing popularity as a platform for server as well as desktop systems. The experiments we have done quantify the scheduling overhead and proportional share allocation accuracy of these schedulers in a real operating system environment under a number of different workloads. Our measurements of kernel scheduler performance are presented in Sections 5.2 to 5.4.

All of our kernel scheduler measurements were performed on a Gateway 2000 E1400 system with a 433 MHz Intel Celeron CPU, 128 MB RAM, and 10 GB hard drive. The system was installed with the Red Hat Linux 6.1 distribution running the Linux version 2.2.12-20 kernel. The measurements were done by using a minimally intrusive tracing facility that logs events at significant points in the application and the operating system code. This is done via a light-weight mechanism that writes timestamped event identifiers into a memory log. The mechanism takes advantage of the high-

resolution clock cycle counter available with the Intel CPU to provide measurement resolution at the granularity of a few nanoseconds. Getting a timestamp simply involved reading the hardware cycle counter register, which could be read from user-level or kernel-level code. We measured the cost of the mechanism on the system to be roughly 70 ns per event.

The kernel scheduler measurements were performed on a fully functional system to represent a realistic system environment. By fully functional, we mean that all experiments were performed with all system functions running and the system connected to the network. At the same time, an effort was made to eliminate variations in the test environment to make the experiments repeatable.

5.1 Simulation Studies

We built a scheduling simulator that we used to evaluate the proportional fairness of VTRR in comparison to two other schedulers, WRR and WFQ. The simulator is a user-space program that measures the service time error, described in Section 2.1, of a scheduler on a set of clients. The simulator takes four inputs, the scheduling algorithm, the number of clients N , the total number of shares S , and the number of client-share combinations. The simulator randomly assigns shares to clients and scales the share values to ensure that they sum to S . It then schedules the clients using the specified algorithm as a real scheduler would, and tracks the resulting service time error. The simulator runs the scheduler until the resulting schedule repeats, then computes the maximum (most positive) and minimum (most negative) service time error across the nonrepeating portion of the schedule for the given set of clients and share assignments. The simulator assumes that all clients are runnable at all times. This process of random share allocation and scheduler simulation is repeated for the specified number of client-share combinations. We then compute an average highest service time error and average lowest service time error for the specified number of client-share combinations to obtain an "average-case" error range.

To measure proportional fairness accuracy, we ran simulations for each scheduling algorithm considered on 40 different combinations of N and S . For each set of (N, S) , we ran 10,000 client-share combinations and determined the resulting average error ranges. The average service time error ranges for VTRR, WRR, and WFQ are shown in Figures 1 and 2.

Figure 1 shows a comparison of the error ranges for VTRR versus WRR, one graph showing the error ranges for VTRR and the other showing the error ranges for WRR. Each graph shows two surfaces plotted on axes of the same scale, representing the maximum and mini-

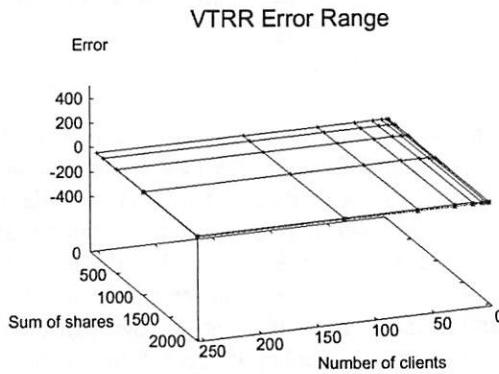
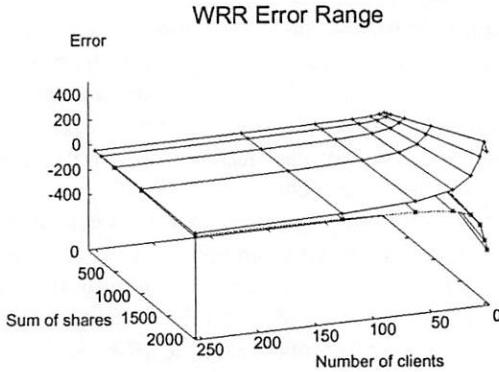


Figure 1: VTRR vs. WRR service time error

mum service time error as a function of N and S . Within the range of values of N and S shown, WRR's error range reaches as low as -398 tu and as high as 479 tu. With the time units expressed in 10 ms jiffies as in Linux, a client under WRR can on average get ahead of its correct CPU time allocation by 4.79 seconds, or behind by 3.98 seconds, which is a substantial amount of service time error. In contrast, Figure 1 shows that VTRR has a much smaller error range than WRR and is much more accurate. Because the error axis is scaled to display the wide range of WRR's error values, it is difficult to even distinguish the two surfaces for VTRR in Figure 1. VTRR's service time error only ranges from -3.8 to 10.6 tu; this can be seen more clearly in Figure 2.

Figure 2 shows a comparison of the error ranges for VTRR versus WFQ, one graph showing the error ranges for VTRR and the other showing the error ranges for WFQ. As in the case in Figure 2, each graph shows two surfaces plotted on axes of the same scale, representing the maximum and minimum service time error as a function of N and S . The VTRR graph in Figure 2 includes the same data as the VTRR graph in Figure 1, but the error axis is scaled more naturally. Within the range of values of N and S shown, WFQ's average error range reaches as low as -1 tu and as high as 2 tu, as opposed

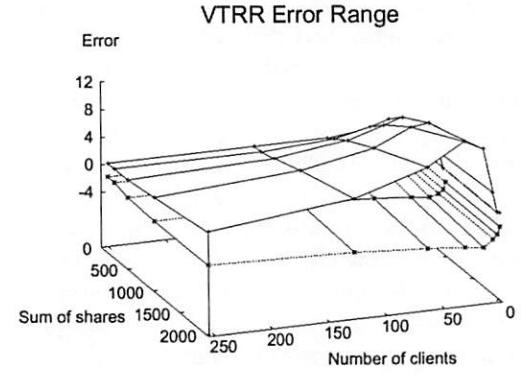
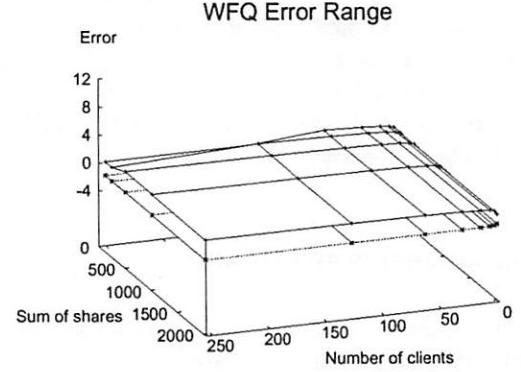


Figure 2: VTRR vs. WFQ service time error

to VTRR's error range from -3.8 to 10.6 tu. The error ranges for WFQ are smaller than VTRR, but the difference between WFQ and VTRR is much smaller than the difference between VTRR and WRR. With the time units expressed in 10 ms jiffies as in Linux, a client under WFQ can on average get ahead of its correct CPU time allocation by 10 ms, or behind by 20 ms, while a client under VTRR can get ahead by 38 ms or behind by 106 ms. In both cases, the service time errors are small. In fact, the service time errors are even below the threshold of delay noticeable by most human beings for response time on interactive applications [27]. Note that another fair queueing algorithm WF^2Q was not simulated, but its error is mathematically bounded [3] between -1 and $+1$ tu, and so would be very similar to WFQ in practice.

The data produced by our simulations confirm that VTRR has fairness properties that are much better than WRR, and nearly as good as WFQ. For the domain of values simulated, the service time error for VTRR falls into an average range almost two orders of magnitude smaller than WRR's error range. While VTRR's error range is not quite as good as WFQ, even the largest error measured, 10.6 tu, would likely be unnoticeable in most applications, given the size of time unit used by most schedulers. Furthermore, we show in Section 5.2 that

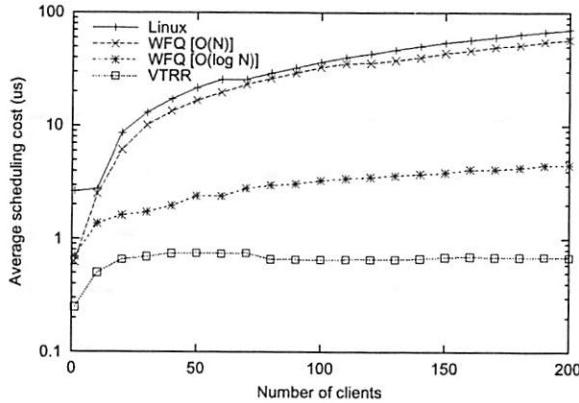


Figure 3: Average scheduling overhead

VTRR provides this degree of accuracy at much lower overhead than WFQ.

5.2 Scheduling Overhead

To evaluate the scheduling overhead of VTRR, we implemented VTRR in the Linux operating system and compared the overhead of our prototype VTRR implementation against the overhead of both the Linux scheduler and a WFQ scheduler. We conducted a series of experiments to quantify how the scheduling overhead for each scheduler varies as the number of clients increases. For this experiment, each client executed a simple micro-benchmark which performed a few operations in a while loop. A control program was used to fork a specified number of clients. Once all clients were runnable, we measured the execution time of each scheduling operation that occurred during a fixed time duration of 30 seconds. This was done by inserting a counter and timestamped event identifiers in the Linux scheduling framework. The measurements required two timestamps for each scheduling decision, so variations of 140 ns are possible due to measurement overhead. We performed these experiments on the standard Linux scheduler, WFQ, and VTRR for 1 client up to 200 clients.

Figure 3 shows the average execution time required by each scheduler to select a client to execute. For this experiment, the particular implementation details of the WFQ scheduler affect the overhead, so we include results from two different implementations of WFQ. In the first, labeled “WFQ [$O(N)$]” the run queue is implemented as a simple linked list which must be searched on every scheduling decision. The second, labeled “WFQ [$O(\log N)$]” uses a heap-based priority queue with $O(\log N)$ insertion time. Most fair queueing-based schedulers are implemented in the first fashion, due to the difficulty of maintaining complex data structures in the kernel. In our implementation, for example, a sep-

arate, fixed-length array was necessary to maintain the heap-based priority queue. If the number of clients ever exceeds the length of the array, a costly array reallocation must be performed. We chose an initial array size large enough to contain more than 200 clients, so this additional cost is not reflected in our measurements.

As shown in Figure 3, the increase in scheduling overhead as the number of clients increases varies a great deal between different schedulers. VTRR has the smallest scheduling overhead. It requires less than 800 ns to select a client to execute and the scheduling overhead is essentially constant for all numbers of clients. In contrast, the overhead for Linux and for $O(N)$ WFQ scheduling grows linearly with the number of clients. The Linux scheduler imposes 100 times more overhead than VTRR when scheduling a mix of 200 clients. In fact, the Linux scheduler still spends almost 10 times as long scheduling a single micro-benchmark client as VTRR does scheduling 200 clients. VTRR outperforms Linux and WFQ even for small numbers of clients because the VTRR scheduling code is simpler and hence runs significantly faster. VTRR performs even better compared to Linux and WFQ for large numbers of clients because it has constant time overhead as opposed to the linear time overhead of the other schedulers.

While $O(\log N)$ WFQ has much smaller overhead than Linux or $O(N)$ WFQ, it still imposes significantly more overhead than VTRR, particularly with large numbers of clients. With 200 clients, $O(\log N)$ WFQ has an overhead more than 6 times that of VTRR. WFQ’s more complex data structures require more time to maintain, and the time required to make a scheduling decision is still dependent on the number of clients, so the overhead would only continue to grow worse as more clients are added. VTRR’s scheduling decisions always take the same amount of time, regardless of the number of clients.

5.3 Microscopic View of Scheduling

Using our prototype VTRR implementation, we conducted a number of experiments to measure the scheduling behavior of the standard Linux scheduler, WFQ, and VTRR at fine time resolutions. We discuss the results of one of the studies in which we ran a 30 second workload of five micro-benchmarks with different proportional sharing parameters. Using VTRR and WFQ, we ran the five micro-benchmarks with shares 1, 2, 3, 4, and 5, respectively. To provide similar proportional sharing behavior using the Linux scheduler, we ran the five micro-benchmarks with user priorities 19, 17, 15, 13, and 11, respectively. This translates to internal priorities used by the scheduler of 1, 3, 5, 7, and 9, respectively. This then translates into the clients running for 20

ms, 40 ms, 60 ms, 80 ms, and 100 ms time quanta, respectively. The smallest time quantum used is the same for all three schedulers. At the very least, the mapping between proportional sharing and user input priorities is non-intuitive in Linux. The scheduling behavior for this workload appears similar across all of the schedulers when viewed at a coarse granularity. The relative resource consumption rates of the micro-benchmarks are virtually identical to their respective shares at a coarse granularity.

We can see more interesting behavior when we view the measurements over a shorter time scale of one second. We show the actual scheduling sequences on each scheduler over this time interval in Figures 4, 5, and 6. These measurements were made by sampling a client's execution from within the client by recording multiple high resolution timestamps each time that a client was executed. We can see that the Linux scheduler does the poorest job of scheduling the clients evenly and predictably. Both WFQ and VTRR do a much better job of scheduling the clients proportionally at a fine granularity. In both cases, there is a clear repeating scheduling pattern every 300 ms.

Linux does not have a perfect repeating pattern because the order in which it schedules clients changes depending on exactly when the scheduler function is called. This is because once Linux selects a client to execute, it does not preempt the client even if its goodness drops below that of other clients. Instead, it runs the client until its counter drops below zero or an interrupt or other scheduling event occurs. If a scheduling event occurs, then Linux will again consider the goodness of all clients, otherwise it does not. Since interrupts can cause a scheduling event and can occur at arbitrary times, the resulting order in which clients are scheduled does not have a repeating pattern. As a result, applications being scheduled using WFQ and VTRR will receive a more even level of CPU service than if they are scheduled using the Linux scheduler.

5.4 Application Workloads

To demonstrate VTRR's efficient proportional sharing of resources on real applications, we briefly describe two of our experiments, one running multimedia applications and the other running virtual machines. We contrast the performance of VTRR versus the standard Linux scheduler and WFQ.

One experiment we performed was to run multiple MPEG audio encoders with different shares on each of the three schedulers. The encoder test was implemented by running five copies of an MPEG audio encoder. The encoder clients were allotted shares of 1, 2, 3, 4, and 5, and were instrumented with time stamp event recorders

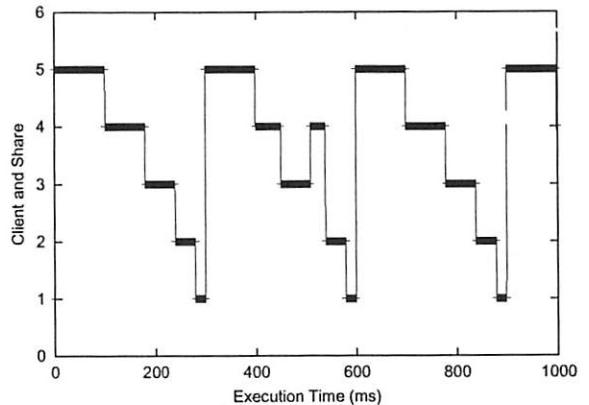


Figure 4: Linux scheduling behavior

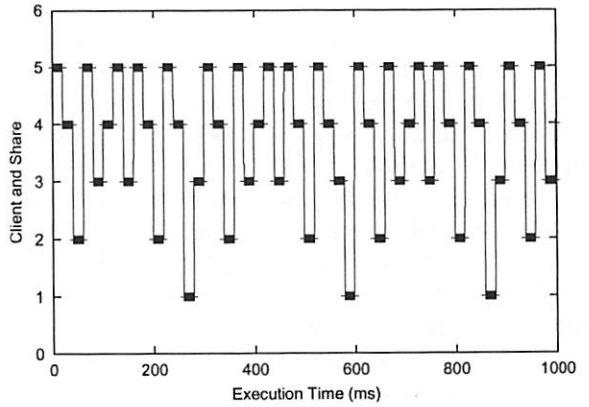


Figure 5: WFQ scheduling behavior

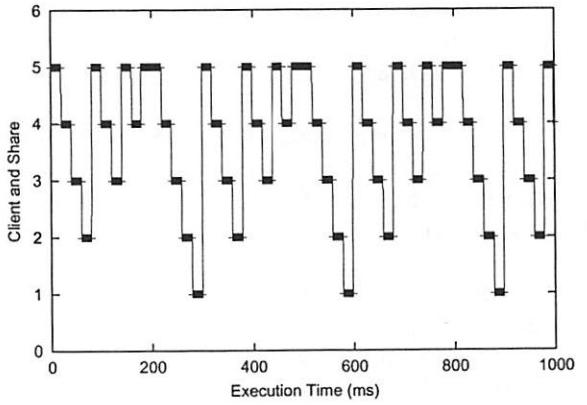


Figure 6: VTRR scheduling behavior

in a manner similar to how we recorded time in our micro-benchmark programs. Each encoder took its input from the same file, but wrote output to its own file. MPEG audio is encoded in chunks called frames, so our instrumented encoder records a timestamp after each frame is encoded, allowing us to easily observe the effect of resource share on single-frame encoding time.

Figures 7, 8, and 9 show the number of frames encoded over time for the Linux default scheduler, WFQ, and VTRR. The Linux scheduler clearly does not provide sharing as fairly as WFQ or VTRR when viewed over a short time interval. The “staircase” effect indicates that CPU resources are provided in bursts, which, for a time-critical task like audio streaming, can mean extra jitter, resulting in delays and dropouts. It can be inferred from the smoother curves of the WFQ and VTRR graphs that WFQ and VTRR scheduling provide fair resource allocation at a much smaller granularity. When analyzed at a fine resolution, we can detect some differences in the proportional sharing behavior of the applications when running under WFQ versus VTRR, but the difference is far smaller than the difference compared with Linux, which is clearly visible. VTRR trades some precision in instantaneous proportional fairness for much lower scheduling overhead.

Schedulers that explicitly support time constraints can do a more effective job than just proportional share schedulers of ensuring that real-time applications can meet their deadlines [24]. However, these real-time schedulers typically require modifying an application in order for the application to make use of scheduler-supported time constraints. For applications that have soft timing constraints but can adapt to the availability of resources, accurate proportional sharing may provide sufficient benefit in some cases without the cost of having to modify the applications.

Another experiment we performed was to run several VMware virtual machines on top a Linux operating system, and then compare the performance of applications within the virtual machines when the virtual machines were scheduled using different schedulers. For this experiment, we ran three virtual machines simultaneously with respective shares of 1, 2, and 3. We then executed a simple timing benchmark within each virtual machine to measure the relative performance of the virtual machine. We were careful to make use of the hardware clock cycle counters in doing these measurements as the standard operating system timing mechanisms within a virtual machine are a poor measure of elapsed time. We conducted the experiment using the standard Linux scheduler, WFQ, and VTRR. The results were similar to the previous experiments, with Linux doing the worst job in terms of evenly distributing CPU cycles, and VTRR and WFQ scheduling providing more comparable schedul-

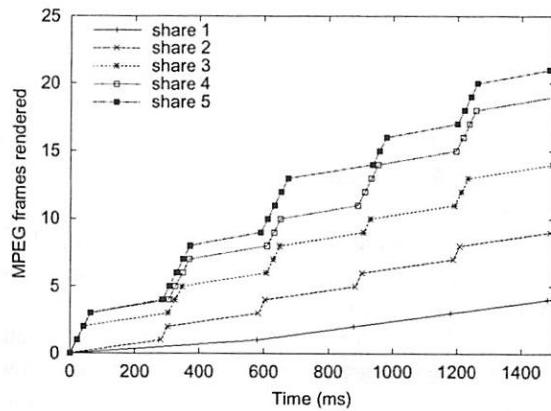


Figure 7: MPEG encoding with Linux

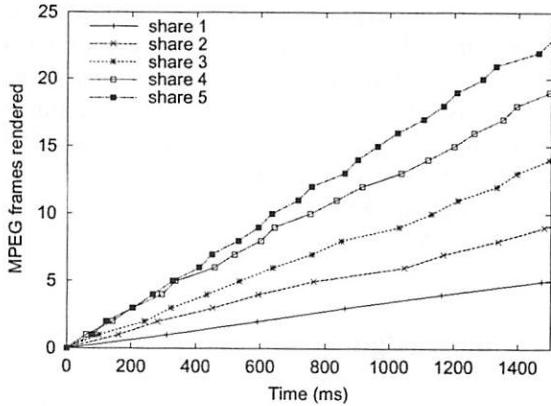


Figure 8: MPEG encoding with WFQ

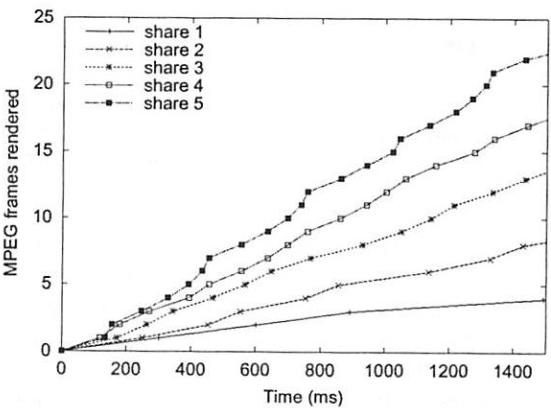


Figure 9: MPEG encoding with VTRR

ing accuracy in proportionally allocating resources.

6 Conclusions and Future Work

We have designed, implemented, and evaluated Virtual-Time Round-Robin scheduling in the Linux operating system. Our experiences with VTRR show that it is simple to implement and easy to integrate into existing commercial operating systems. We have measured the performance of our Linux implementation and demonstrated that VTRR combines the benefits of accurate proportional share resource management with very low overhead. Our results show that VTRR scheduling overhead is constant, even for large numbers of clients. Despite the popularity of the Linux operating system, our results also show that the standard Linux scheduler suffers from $O(N)$ scheduling overhead and performs much worse than VTRR, especially for larger workloads.

VTRR's ability to provide low overhead proportional share resource allocation make it a particularly promising solution for managing resources in large-scale server systems. Since these systems are typically multiprocessor machines, we are continuing our evaluation of VTRR in a multiprocessor context to demonstrate its effectiveness in supporting large numbers of users and applications in these systems.

7 Acknowledgments

We thank the anonymous referees for their helpful comments on earlier drafts of this paper. This work was supported in part by NSF grant EIA-0071954 and an NSF CAREER Award.

References

- [1] G. Banga, P. Druschel, and J. Mogul, "Resource Containers: A New Facility for Resource Management in Server Systems," in *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, USENIX, Berkeley, CA, Feb. 22–25 1999, pp. 45–58.
- [2] M. Beck, H. Bohme, M. Dziadzka, and U. Kunitz, *Linux Kernel Internals*. Reading, MA: Addison-Wesley, 2nd ed., 1998.
- [3] J. Bennett and H. Zhang, "WF²Q: Worst-case Fair Weighted Fair Queueing," in *Proceedings of INFOCOM '96*, San Francisco, CA, Mar. 1996.
- [4] G. Bollella and K. Jeffay, "Support for Real-Time Computing Within General Purpose Operating Systems: Supporting Co-resident Operating Systems," in *Proceedings of the Real-Time Technology and Applications Symposium*, IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995, pp. 4–14.
- [5] R. Bryant and B. Hartner, "Java Technology, Threads, and Scheduling in Linux," IBM developerWorks Library Paper, IBM Linux Technology Center, Jan. 2000.
- [6] G. Coulson, A. Campbell, P. Robin, G. Blair, M. Papathomas, and D. Hutchinson, "Design of a QoS Controlled ATM Based Communication System in Chorus," in *IEEE Journal of Selected Areas in Communications (JSAC)*, 13(4), May 1995, pp. 686–699.
- [7] H. Custer, *Inside Windows NT*. Redmond, WA, USA: Microsoft Press, 1993.
- [8] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *Proceedings of ACM SIGCOMM '89*, Austin, TX, Sept. 1989, pp. 1–12.
- [9] K. Duda and D. Cheriton, "Borrowed-Virtual-Time (BVT) Scheduling: Supporting Latency-Sensitive Threads in a General-Purpose Scheduler," in *Proceedings of the 17th Symposium on Operating Systems Principles*, ACM Press, New York, Dec. 1999, pp. 261–276.
- [10] R. Essick, "An Event-Based Fair Share Scheduler," in *Proceedings of the Winter 1990 USENIX Conference*, USENIX, Berkeley, CA, USA, Jan. 1990, pp. 147–162.
- [11] S. Evans, K. Clarke, D. Singleton, and B. Smaalders, "Optimizing Unix Resource Scheduling for User Interaction," in *1993 Summer Usenix*, USENIX, June 1993, pp. 205–218.
- [12] E. Gafni and D. Bertsekas, "Dynamic Control of Session Input Rates in Communication Networks," in *IEEE Transactions on Automatic Control*, 29(10), 1984, pp. 1009–1016.
- [13] D. Golub, "Operating System Support for Coexistence of Real-Time and Conventional Scheduling," Tech. Rep. CMU-CS-94-212, School of Computer Science, Carnegie Mellon University, Nov. 1994.
- [14] P. Goyal, X. Guo, and H. Vin, "A Hierarchical CPU Scheduler for Multimedia Operating System," in *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, USENIX, Berkeley, CA, Oct. 1996, pp. 107–121.
- [15] E. Hahne and R. Gallager, "Round Robin Scheduling for Fair Flow Control in Data Communication Networks," Tech. Rep. LIDS-TH-1631, Laboratory for Information and Decision Systems, Massachusetts Institute of Technology, Dec. 1986.
- [16] G. Henry, "The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal*, 63(8), Oct. 1984, pp. 1845–1857.
- [17] M. Jones, D. Roșu, and M. Roșu, "CPU Reservations and Time Constraints: Efficient, Predictable Scheduling of Independent Activities," in *Proceedings of the 16th Symposium on Operating Systems Principles*, ACM Press, New York, Oct. 1997, pp. 198–211.
- [18] J. Kay and P. Lauder, "A Fair Share Scheduler," *Communications of the ACM*, 31(1), Jan. 1988, pp. 44–55.
- [19] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications*. New York: John Wiley & Sons, 1976.
- [20] I. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," in *Proceedings of the Real-Time Systems Symposium - 1989*, IEEE Computer Society Press, Santa Monica, California, USA, Dec. 1989, pp. 166–171.
- [21] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM*, 20(1), Jan. 1973, pp. 46–61.
- [22] C. Locke, *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 1986.

- [23] C. Mercer, S. Savage, and H. Tokuda, "Processor Capacity Reserves: Operating System Support for Multimedia Applications," in *Proceedings of the International Conference on Multimedia Computing and Systems*, IEEE Computer Society Press, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1994, pp. 90–99.
- [24] J. Nich and M. Lam, "The Design, Implementation and Evaluation of SMART: A Scheduler for Multimedia Applications," in *Proceedings of the 16th Symposium on Operating Systems Principles*, 31(5), ACM Press, New York, Oct. 5–8 1997, pp. 184–197.
- [25] A. Parekh and R. Gallager, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks: The Single-Node Case," *IEEE/ACM Transactions on Networking*, 1(3), June 1993, pp. 344–357.
- [26] K. Ramakrishnan, D. Chiu, and R. Jain, "Congestion Avoidance in Computer Networks with a Connectionless Network Layer, Part IV: A Selective Binary Feedback Scheme for General Topologies," Tech. Rep. DEC-TR-510, DEC, Nov. 1987.
- [27] B. Shneiderman, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, MA: Addison-Wesley, 2nd ed., 1992.
- [28] M. Shreedhar and G. Varghese, "Efficient Fair Queueing Using Deficit Round-Robin," in *Proceedings of ACM SIGCOMM '95*, 4(3), Sept. 1995, pp. 231–242.
- [29] A. Silberschatz and P. Galvin, *Operating System Concepts*. Reading, MA, USA: Addison-Wesley, 5th ed., 1998.
- [30] I. Stoica, H. Abdel-Wahab, and K. Jeffay, "On the Duality between Resource Reservation and Proportional Share Resource Allocation," in *Multimedia Computing and Networking Proceedings, SPIE Proceedings Series*, 3020, Feb. 1997, pp. 207–214.
- [31] "UNIX System V Release 4 Internals Student Guide, Vol. I, Unit 2.4.2." AT&T, 1990.
- [32] R. Tijdeman, "The Chairman Assignment Problem," *Discrete Mathematics*, 32, 1980, pp. 323–330.
- [33] C. Waldspurger, *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995.
- [34] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switched Networks," in *ACM Transactions on Computer Systems*, 9(2), May 1991, pp. 101–125.

A toolkit for user-level file systems

David Mazières

Department of Computer Science, NYU

`dm@cs.nyu.edu`

Abstract

This paper describes a C++ toolkit for easily extending the Unix file system. The toolkit exposes the NFS interface, allowing new file systems to be implemented portably at user level. A number of programs have implemented portable, user-level file systems. However, they have been plagued by low-performance, deadlock, restrictions on file system structure, and the need to reboot after software errors. The toolkit makes it easy to avoid the vast majority of these problems. Moreover, the toolkit also supports user-level access to existing file systems through the NFS interface—a heretofore rarely employed technique. NFS gives software an asynchronous, low-level interface to the file system that can greatly benefit the performance, security, and scalability of certain applications. The toolkit uses a new asynchronous I/O library that makes it tractable to build large, event-driven programs that never block.

1 Introduction

Many applications could reap a number of benefits from a richer, more portable file system interface than that of Unix. This paper describes a toolkit for portably extending the Unix file system—both facilitating the creation of new file systems and granting access to existing ones through a more powerful interface. The toolkit exploits both the client and server sides of the ubiquitous Sun Network File System [15]. It lets the file system developer build a new file system by emulating an NFS server. It also lets application writers replace file system calls with networking calls, permitting lower-level manipulation of files and working around such limitations as the maximum number of open files and the synchrony of many operations.

We used the toolkit to build the SFS distributed file system [13], and thus refer to it as the SFS file system development toolkit. SFS is relied upon for daily use by several people, and thus shows by example that one can build production-quality NFS loopback servers. In addition, other users have picked up the toolkit and built functioning Unix file systems in a matter of a week. We

have even used the toolkit for class projects, allowing students to build real, functioning Unix file systems.

Developing new Unix file systems has long been a difficult task. The internal kernel API for file systems varies significantly between versions of the operating system, making portability nearly impossible. The locking discipline on file system data structures is hair-raising for the non-expert. Moreover, developing in-kernel file systems has all the complications of writing kernel code. Bugs can trigger a lengthy crash and reboot cycle, while kernel debugging facilities are generally less powerful than those for ordinary user code.

At the same time, many applications could benefit from an interface to existing file systems other than POSIX. For example, non-blocking network I/O permits highly efficient software in many situations, but any synchronous disk I/O blocks such software, reducing its throughput. Some operating systems offer asynchronous file I/O through the POSIX *aio* routines, but *aio* is only for reading and writing files—it doesn't allow files to be opened and created asynchronously, or directories to be read.

Another shortcoming of the Unix file system interface is that it foments a class of security holes known as time of check to time of use, or TOCTTOU, bugs [2]. Many conceptually simple tasks are actually quite difficult to implement correctly in privileged software—for instance, removing a file without traversing a symbolic link, or opening a file on condition that it be accessible to a less privileged user. As a result, programmers often leave race conditions that attackers can exploit to gain greater privilege.

The next section summarizes related work. Section 3 describes the issues involved in building an NFS loopback server. Section 4 explains how the SFS toolkit facilitates the construction of loopback servers. Section 5 discusses loopback clients. Section 6 describes applications of the toolkit and discusses performance. Finally, Section 7 concludes.

2 Related work

A number of file system projects have been implemented as NFS loopback servers. Perhaps the first example is the Sun *automount* daemon [5]—a daemon that mounts remote NFS file systems on-demand when their pathnames are referenced. Neither *automount* nor a later, more advanced automounter, *amd* [14], were able to mount file systems in place to turn a pathname referenced by a user into a mount point on-the-fly. Instead, they took the approach of creating mount points outside of the directory served by the loopback server, and redirecting file accesses using symbolic links. Thus, for example, *amd* might be a loopback server for directory `/home`. When it sees an access to the path `/home/am2`, it will mount the corresponding file system somewhere else, say on `/a/amsterdam/u2`, then produce a symbolic link, `/home/am2 → /a/amsterdam/u2`. This symbolic link scheme complicates life for users. For this and other reasons, Solaris and Linux pushed part of the automounter back into the kernel. The SFS toolkit shows they needn’t have done so for mounting in place, one can in fact implement a proper automounter as a loopback server.

Another problem with previous loopback automounters is that one unavailable server can impede access to other, functioning servers. In the example from the previous paragraph, suppose the user accesses `/home/am2` but the corresponding server is unavailable. It may take *amd* tens of seconds to realize the server is unavailable. During this time, *amd* delays responding to an NFS request for file `am2` in `/home`. While the the lookup is pending, the kernel’s NFS client will lock the `/home` directory, preventing access to all other names in the directory as well.

Loopback servers have been used for purposes other than automounting. CFS [3] is a cryptographic file system implemented as an NFS loopback server. Unfortunately, CFS suffers from deadlock. It predicates the completion of loopback NFS write calls on writes through the file system interface, which, as discussed later, leads to deadlock. The Alex ftp file system [7] is implemented using NFS. However Alex is read-only, which avoids any deadlock problems. Numerous other file systems are constructed as NFS loopback servers, including the semantic file system [9] and the Byzantine fault-tolerant file system [6]. The SFS toolkit makes it considerably easier to build such loopback servers than before. It also helps avoid many of the problems previous loopback servers have had. Finally, it supports NFS loopback clients, which have advantages discussed later on.

New file systems can also be implemented by replacing system shared libraries or even intercepting all of a process’s system calls, as the UFO system does [1]. Both

methods are appealing because they can be implemented by a completely unprivileged user. Unfortunately, it is hard to implement complete file system semantics using these methods (for instance, you can’t hand off a file descriptor using `sendmsg()`). Both methods also fail in some cases. Shared libraries don’t work with statically linked applications, and neither approach works with setuid utilities such as *lpr*. Moreover, having different namespaces for different processes can cause confusion, at least on operating systems that don’t normally support this.

FiST [19] is a language for generating stackable file systems, in the spirit of Ficus [11]. FiST can output code for three operating systems—Solaris, Linux, and FreeBSD—giving the user some amount of portability. FiST outputs kernel code, giving it the advantages and disadvantages of being in the operating system. FiST’s biggest contributions are really the programming language and the stackability, which allow simple and elegant code to do powerful things. That is somewhat orthogonal to the SFS toolkit’s goals of allowing file systems at user level (though FiST is somewhat tied to the VFS layer—it couldn’t unfortunately be ported to the SFS toolkit very easily). Aside from its elegant language, the big trade-off between FiST and the SFS toolkit is performance vs. portability and ease of debugging. Loopback servers will run on virtually any operating system, while FiST file systems will likely offer better performance.

Finally, several kernel device drivers allow user-level programs to implement file systems using an interface other than NFS. The now defunct UserFS [8] exports an interface similar to the kernel’s VFS layer to user-level programs. UserFS was very general, but only ran on Linux. Arla [17], an AFS client implementation, contains a device, `xfs`, that lets user-level programs implement a file system by sending messages through `/dev/xfs0`. Arla’s protocol is well-suited to network file systems that perform whole file caching, but not as general-purpose as UserFS. Arla runs on six operating systems, making `xfs`-based file systems portable. However, users must first install `xfs`. Similarly, the Coda file system [12] uses a device driver `/dev/cfs0`.

3 NFS loopback server issues

NFS loopback servers allow one to implement a new file system portably, at user-level, through the NFS protocol rather than some operating-system-specific kernel-internal API (e.g., the VFS layer). Figure 1 shows the architecture of an NFS loopback server. An application accesses files using system calls. The operating system’s NFS client implements the calls by sending NFS requests

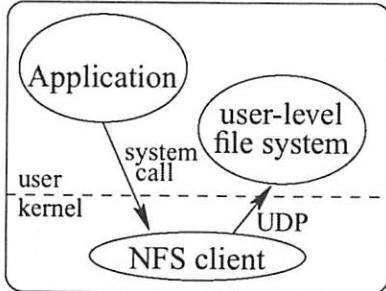


Figure 1: A user-level NFS loopback server

to the user-level server. The server, though treated by the kernel’s NFS code as if it were on a separate machine, actually runs on the same machine as the applications. It responds to NFS requests and implements a file system using only standard, portable networking calls.

3.1 Complications of NFS loopback servers

Making an NFS loopback server perform well poses a few challenges. First, because it operates at user-level, a loopback server inevitably imposes additional context switches on applications. There is no direct remedy for the situation. Instead, the loopback file system implementer must compensate by designing the rest of the system for high performance.

Fortunately for loopback servers, people are willing to use file systems that do not perform optimally (NFS itself being one example). Thus, a file system offering new functionality can be useful as long as its performance is not unacceptably slow. Moreover, loopback servers can exploit ideas from the file system literature. SFS, for instance, manages to maintain performance competitive with NFS by using leases [10] for more aggressive attribute and permission caching. An in-kernel implementation could have delivered far better performance, but the current SFS is a useful system because of its enhanced security.

Another performance challenge is that loopback servers must handle multiple requests in parallel. Otherwise, if, for instance, a server waits for a request of its own over the network or waits for a disk read, multiple requests will not overlap their latencies and the overall throughput of the system will suffer.

Worse yet, any blocking operation performed by an NFS loopback server has the potential for deadlock. This is because of typical kernel buffer allocation strategy. On many BSD-derived Unixes, when the kernel runs out of buffers, the buffer allocation function can pick some dirty buffer to recycle and block until that particular buffer has

been cleaned. If cleaning that buffer requires calling into the loopback server and the loopback server is waiting for the blocked kernel thread, then deadlock will ensue.

To avoid deadlock, an NFS loopback server must never block under any circumstances. Any file I/O within a loopback server is obviously strictly prohibited. However, the server must avoid page faults, too. Even on operating systems that rigidly partition file cache and program memory, a page fault needs a `struct buf` to pass to the disk driver. Allocating the structure may in turn require that some file buffer be cleaned. In the end, a mere debugging `printf` can deadlock a system; it may fill the queue of a pseudo-terminal handled by a remote login daemon that has suffered a page fault (an occurrence observed by the author). A large piece of software that never blocks requires fundamentally different abstractions from most other software. Simply using an in-kernel threads package to handle concurrent NFS requests at user level isn’t good enough, as the thread that blocks may be the one cleaning the buffer everyone is waiting for.

NFS loopback servers are further complicated by the kernel NFS client’s internal locking. When an NFS request takes too long to complete, the client retransmits it. After some number of retransmissions, the client concludes that the server or network has gone down. To avoid flooding the server with retransmissions, the client locks the mount point, blocking any further requests, and periodically retransmitting only the original, slow request. This means that a single “slow” file on an NFS loopback server can block access to other files from the same server.

Another issue faced by loopback servers is that a lot of software (e.g., Unix implementations of the ANSI C `getcwd()` function) requires every file on a system to have a unique `(st_dev, st_ino)` pair. `st_dev` and `st_ino` are fields returned by the POSIX `stat()` function. Historically, `st_dev` was a number designating a device or disk partition, while `st_ino` corresponded to a file within that disk partition. Even though the NFS protocol has a field equivalent to `st_dev`, that field is ignored by Unix NFS clients. Instead, all files under a given NFS mount point are assigned a single `st_dev` value, made up by the kernel. Thus, when stitching together files from various sources, a loopback server must ensure that all `st_ino` fields are unique for a given mount point.

A loopback server can avoid some of the problems of slow files and `st_ino` uniqueness by using multiple mount points—effectively emulating several NFS servers. One often would like to create these mount points on-the-fly—for instance to “automount” remote servers as the user references them. Doing so is non-trivial because of vnode locking on file name lookups.

While the NFS client is looking up a file name, one cannot in parallel access the same name to create a new mount point. This drove previous NFS loopback automounters to create mount points outside of the loopback file system and serve only symbolic links through the loopback mount.

As user-level software, NFS loopback servers are easier to debug than kernel software. However, a buggy loopback server can still hang a machine and require a reboot. When a loopback server crashes, any reference to the loopback file system will block. Hung processes pile up, keeping the file system in use and on many operating systems preventing unmounting. Even the *umount* command itself sometimes does things that require an NFS RPC, making it impossible to clean up the mess without a reboot. If a loopback file system uses multiple mount points, the situation is even worse, as there is no way to traverse higher level directories to unmount the lower-level mount points.

In summary, while NFS loopback servers offer a promising approach to portable file system development, a number of obstacles must be overcome to build them successfully. The goal of the SFS file system development toolkit is to tackle these problems and make it easy for people to develop new file systems.

4 NFS loopback server toolkit

This section describes how the SFS toolkit supports building robust user-level loopback servers. The toolkit has several components, illustrated in Figure 2. *nfsmounter* is a daemon that creates and deletes mount points. It is the only part of the SFS client that needs to run as root, and the only part of the system that must function properly to prevent a machine from getting wedged. The SFS automounter daemon creates mount points dynamically as users access them. Finally, a collection of libraries and a novel RPC compiler simplify the task of implementing entirely non-blocking NFS loopback servers.

4.1 Basic API

The basic API of the toolkit is effectively the NFS 3 protocol [4]. The server allocates an *nfsserv* object, which might, for example, be bound to a UDP socket. The server hands this object a dispatch function. The object then calls the dispatch function with NFS 3 RPCs. The dispatch function is asynchronous. It receives an argument of type pointer to *nfscall*, and it returns nothing. To reply to an NFS RPC, the server calls the *reply* method of the *nfscall* object. This needn't happen before the dispatch routine returns, however. The *nfscall*

can be stored away until some other asynchronous event completes.

4.2 The *nfsmounter* daemon

The purpose of *nfsmounter* is to clean up the mess when other parts of the system fail. This saves the loopback file system developer from having to reboot the machine, even if something goes horribly wrong with his loopback server. *nfsmounter* runs as root and calls the *mount* and *umount* (or *umount*) system calls at the request of other processes. However, it aggressively distrusts these processes. Its interface is carefully crafted to ensure that *nfsmounter* can take over and assume control of a loopback mount whenever necessary.

nfsmounter communicates with other daemons through Unix domain sockets. To create a new NFS mount point, a daemon first creates a UDP socket over which to speak the NFS protocol. The daemon then passes this socket and the desired pathname for the mount point to *nfsmounter* (using Unix domain socket facilities for passing file descriptors across processes). *nfsmounter*, acting as an NFS client to existing loopback mounts, then probes the structure of any loopback file systems traversed down to the requested mount point. Finally, *nfsmounter* performs the actual mount system call and returns the result to the invoking daemon.

After performing a mount, *nfsmounter* holds onto the UDP socket of the NFS loopback server. It also remembers enough structure of traversed file systems to recreate any directories used as mount points. If a loopback server crashes, *nfsmounter* immediately detects this by receiving an end-of-file on the Unix domain socket connected to the server. *nfsmounter* then takes over any UDP sockets used by the crashed server, and begins serving the skeletal portions of the file system required to clean up underlying mount points. Requests to other parts of the file system return stale file handle errors, helping ensure most programs accessing the crashed file system exit quickly with an error, rather than hanging on a file access and therefore preventing the file system from being unmounted.

nfsmounter was built early in the development of SFS. After that point, we were able to continue development of SFS without any dedicated “crash boxes.” No matter what bugs cropped up in the rest of SFS, we rarely needed a reboot. This mirrors the experience of students, who have used the toolkit for class projects without ever knowing the pain that loopback server development used to cause.

On occasion, of course, we have turned up bugs in kernel NFS implementations. We have suffered many kernel panics trying to understand these problems, but, strictly

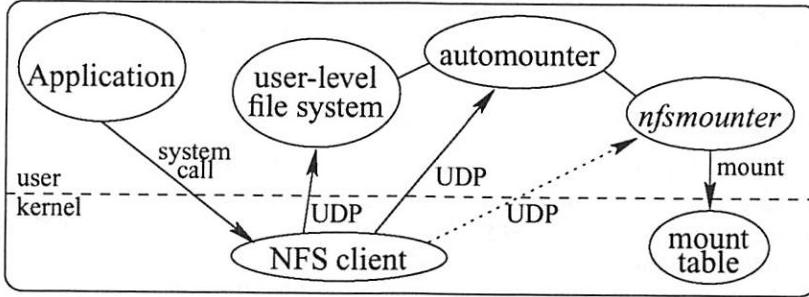


Figure 2: Architecture of the user-level file system toolkit

speaking, that part of the work qualifies as kernel development, not user-level server development.

4.3 Automounting in place

The SFS automounter shows that loopback automounters can mount file systems in place, even though no previous loopback automounter has managed to do so. SFS consists of a top level directory, `/sfs`, served by an automounter process, and a number of subdirectories of `/sfs` served by separate loopback servers. Subdirectories of `/sfs` are created on-demand when users access the directory names. Since subdirectories of `/sfs` are handled by separate loopback servers, they must be separate mount points.

The kernel's vnode locking strategy complicates the task of creating mount points on-demand. More specifically, when a user references the name of an as-yet-unknown mount point in `/sfs`, the kernel generates an NFS LOOKUP RPC. The automounter cannot immediately reply to this RPC, because it must first create a mount point. On the other hand, creating a mount point requires a `mount` system call during which the kernel again looks up the same pathname. The client NFS implementation will already have locked the `/sfs` directory during the first LOOKUP RPC. Thus the lookup within the `mount` call will hang.

Worse yet, the SFS automounter cannot always immediately create a requested mount point. It must validate the name of the directory, which involves a DNS lookup and various other network I/O. Validating a directory name can take a long time, particularly if a DNS server is down. The time can be sufficient to drive the NFS client into retransmission and have it lock the mount point, blocking all requests to `/sfs`. Thus, the automounter cannot sit on any LOOKUP request for a name in `/sfs`. It must reply immediately.

The SFS automounter employs two tricks to achieve what previous loopback automounters could not. First, it tags `nfsmounter`, the process that actually makes the mount system calls, with a reserved group ID (an idea

first introduced by HLFSD [18]). By examining the credentials on NFS RPCs, then, the automounter can differentiate NFS calls made on behalf of `nfsmounter` from those issued for other processes. Second, the automounter creates a number of special ".mnt" mount points on directories with names of the form `/sfs/.mnt/0/`, `/sfs/.mnt/1/`, The automounter never delays a response to a LOOKUP RPC in the `/sfs` directory. Instead, it returns a symbolic link redirecting the user to another symbolic link in one of the .mnt mount points. There it delays the result of a READLINK RPC. Because the delayed readlink takes place under a dedicated mount point, however, no other file accesses are affected.

Meanwhile, as the user's process awaits a READLINK reply under `/sfs/.mnt/n`, the automounter actually mounts the remote file system under `/sfs`. Because `nfsmounter`'s NFS RPCs are tagged with a reserved group ID, the automounter responds differently to them—giving `nfsmounter` a different view of the file system from the user's. While users referencing the pathname in `/sfs` see a symbolic link to `/sfs/.mnt/...`, `nfsmounter` sees an ordinary directory on which it can mount the remote file system. Once the mount succeeds, the automounter lets the user see the directory, and responds to the pending READLINK RPC redirecting the user to the original pathname in `/sfs` which has now become a directory.

A final problem faced by automounters is that the commonly used `getcwd()` library function performs an `Istat` system call on every entry of a directory containing mount points, such as `/sfs`. Thus, if any of the loopback servers mounted on immediate subdirectories of `/sfs` become unresponsive, `getcwd()` might hang, even when run from within a working file system. Since loopback servers may depend on networked resources that become transiently unavailable, a loopback server may well need to become unavailable. When this happens, the loopback server notifies the automounter, and the automounter returns temporary errors to any process attempting to access the problematic mount point (or rather, to any pro-

cess except *nfsmounter*, so that unavailable file systems can still be unmounted).

4.4 Asynchronous I/O library

Traditional I/O abstractions and interfaces are ill-suited to completely non-blocking programming of the sort required for NFS loopback servers. Thus, the SFS file system development toolkit contains a new C++ non-blocking I/O library, *libasync*, to help write programs that avoid any potentially blocking operations. When a function cannot complete immediately, it registers a callback with *libasync*, to be invoked when a particular asynchronous event occurs. At its core, *libasync* supports callbacks when file descriptors become ready for I/O, when child processes exit, when a process receives signals, and when the clock passes a particular time. A central dispatch loop polls for such events to occur through the system call *select*—the only blocking system call a loopback server ever makes.

Two complications arise from this style of event-driven programming in a language like C or C++. First, in languages that do not support closures, it can be inconvenient to bundle up the necessary state one must preserve to finish an operation in a callback. Second, when an asynchronous library function takes a callback and buffer as input and allocates memory for its results, the function’s type signature does not make clear which code is responsible for freeing what memory when. Both complications easily lead to programming errors, as we learned bitterly in the first implementation of SFS which we entirely scrapped.

libasync makes asynchronous library interfaces less error-prone through aggressive use of C++ templates. A heavily overloaded template function, *wrap*, produces callback objects through a technique much like function currying: *wrap* bundles up a function pointer and some initial arguments to pass the function, and it returns a function object taking the function’s remaining arguments. In other words, given a function:

```
res_t function (a1_t, a2_t, a3_t);
```

a call to *wrap* (*function*, *a1*, *a2*) produces a function object with type signature:

```
res_t callback (a3_t);
```

This *wrap* mechanism permits convenient bundling of code and data into callback objects in a type-safe way. Though the example shows the wrapping of a simple function, *wrap* can also bundle an object and method pointer with arguments. *wrap* handles functions and arguments of any type, with no need to declare the combination of types ahead of time. The maximum number of

```
class foo : public bar {
    /* ... */
};

void
function ()
{
    ref<foo> f = new refcounted<foo>
        /* constructor arguments */;

    ptr<bar> b = f;
    f = new refcounted<foo>
        /* constructor arguments */;
    b = NULL;
}
```

Figure 3: Example usage of reference-counted pointers

arguments is determined by a parameter in a perl script that actually generates the code for *wrap*.

To avoid the programming burden of tracking which of a caller and callee is responsible for freeing dynamically allocated memory, *libasync* also supports reference-counted garbage collection. Two template types offer reference-counted pointers to objects of type T—*ptr*<T> and *ref*<T>. *ptr* and *ref* behave identically and can be assigned to each other, except that a *ref* cannot be NULL. One can allocate a reference-counted version of any type with the template type *refcounted*<T>, which takes the same constructor arguments as type T. Figure 3 shows an example use of reference-counted garbage collection. Because reference-counted garbage collection deletes objects as soon as they are no longer needed, one can also rely on destructors of reference-counted objects to release resources more precious than memory, such as open file descriptors.

libasync contains a number of support routines built on top of the core callbacks. It has asynchronous file handles for input and formatted output, an asynchronous DNS resolver, and asynchronous TCP connection establishment. All were implemented from scratch to use *libasync*’s event dispatcher, callbacks, and reference counting. *libasync* also supplies helpful building blocks for objects that accumulate data and must deal with short writes (when no buffer space is available in the kernel). Finally, it supports asynchronous logging of messages to the terminal or system log.

4.5 Asynchronous RPC library and compiler

The SFS toolkit also supplies an asynchronous RPC library, *libarpc*, built on top of *libasync*, and a new RPC

compiler, *rpcc*. *rpcc* compiles Sun XDR data structures into C++ data structures. Rather than directly output code for serializing the structures, however, *rpcc* uses templates and function overloading to produce a generic way of traversing data structures at compile time. This allows one to write concise code that actually compiles to a number of functions, one for each NFS data type. Serialization of data in RPC calls is but one application of this traversal mechanism. The ability to traverse NFS data structures automatically turned out to be useful in a number of other situations.

As an example, one of the loopback servers constituting the client side of SFS uses a protocol very similar to NFS for communicating with remote SFS servers. The only differences is that the SFS protocol has more aggressive file attribute caching and lets the server call back to the client to invalidate attributes. Rather than manually extract attribute information from the return structures of 21 different NFS RPCs, the SFS client uses the RPC library to traverse the data structures and extract attributes automatically. While the compiled output consists of numerous functions, most of these are C++ template instantiations automatically generated by the compiler. The source needs only a few functions to overload the traversal's behavior on attribute structures. Moreover, any bug in the source will likely break all 21 NFS functions.

4.6 Stackable NFS manipulators

An SFS support library, *libfsfsmisc*, provides stackable NFS manipulators. Manipulators take one *nfsserv* object and produce a different one, manipulating any calls from and replies to the original *nfsserv* object. A loopback NFS server starts with an initial *nfsserv* object, generally *nfsserv_udp* which accepts NFS calls from a UDP socket. The server can then push a bunch of manipulators onto this *nfsserv*. For example, over the course of developing SFS we stumbled across a number of bugs that caused panics in NFS client implementations. We developed an NFS manipulator, *nfsserv_fixup*, that works around these bugs. SFS's loopback servers push *nfsserv_fixup* onto their NFS manipulator stack, and then don't worry about the specifics of any kernel bugs. If we discover another bug to work around, we need only put the workaround in a single place to fix all loopback servers.

Another NFS manipulator is a demultiplexer that breaks a single stream of NFS requests into multiple streams. This allows a single UDP socket to be used as the server side for multiple NFS mount points. The demultiplexer works by tagging all NFS file handles with the number of the mount point they belong to. Though file handles are scattered throughout the NFS call and re-

turn types, the tagging was simple to implement using the traversal feature of the RPC compiler.

4.7 Miscellaneous features

The SFS toolkit has several other features. It supplies a small, user-level module, *mallock.o*, that loopback servers must link against to avoid paging. On systems supporting the *mlockall()* system call, this is easily accomplished. On other systems, *mallock.o* manually pins the text and data segments and replaces the *malloc()* library function with a routine that always returns pinned memory.

Finally, the SFS toolkit contains a number of debugging features, including aggressive memory checking, type checking for accesses to RPC union structures, and easily toggleable tracing and pretty-printing of RPC traffic. Pretty-printing of RPC traffic in particular has proven an almost unbeatable debugging tool. Each NFS RPC typically involves a limited amount of computation. Moreover, separate RPC calls are relatively independent of each other, making most problems easily reproducible. When a bug occurs, we turn on RPC tracing, locate the RPC on which the server is returning a problematic reply, and set a conditional breakpoint to trigger under the same conditions. Once in the debugger, it is generally just a matter of stepping through a few functions to understand how we arrive from a valid request to an invalid reply.

Despite the unorthodox structure of non-blocking daemons, the SFS libraries have made SFS's 60,000+ lines of code (including the toolkit and all daemons) quite manageable. The trickiest bugs we have hit were in NFS implementations. At least the SFS toolkit's tracing facilities let us quickly verify that SFS was behaving correctly and pin the blame on the kernel.

4.8 Limitations of NFS loopback servers

Despite the benefits of NFS loopback servers and their tractability given the SFS toolkit, there are two serious drawbacks that must be mentioned. First, the NFS 2 and 3 protocols do not convey file closes to the server. There are many reasons why a file system implementor might wish to know when files are closed. We have implemented a close simulator as an NFS manipulator, but it cannot be 100% accurate and is thus not suitable for all needs. The NFS 4 protocol [16] does have file closes, which will solve this problem if NFS 4 is deployed.

The other limitation is that, because of the potential risk of deadlock in loopback servers, one can never predicate the completion of an NFS write on that of a write issued to local disk. Loopback servers can access the

local disk, provided they do so asynchronously. *libasync* offers support for doing so using helper processes, or one can do so as an NFS loopback client. Thus, one can build a loopback server that talks to a remote server and keeps a cache on the local disk. However, the loopback server must return from an NFS write once the corresponding remote operation has gone through; it cannot wait for the write to go through in the local cache. Thus, techniques that rely on stable, crash-recoverable writes to local disk, such as those for disconnected operation in CODA [12], cannot easily be implemented in loopback servers; one would need to use raw disk partitions.

5 NFS loopback clients

In addition to implementing loopback servers, *libarpc* allows applications to behave as NFS clients, making them loopback clients. An NFS loopback client accesses the local hard disk by talking to an in-kernel NFS server, rather than using the standard POSIX open/close/read/write system call interface. Loopback clients have none of the disadvantages of loopback servers. In fact, a loopback client can still access the local file system through system calls. NFS simply offers a lower-level, asynchronous alternative from which some aggressive applications can benefit.

The SFS server software is actually implemented as an NFS loopback client using *libarpc*. It reaps a number of benefits from this architecture. The first is performance. Using asynchronous socket I/O, the loopback client can have many parallel disk operations outstanding simultaneously. This in turn allows the operating system to achieve better disk arm scheduling and get higher throughput from the disk. Though POSIX does offer optional *aio* system calls for asynchronous file I/O, the *aio* routines only operate on open files. Thus, without the NFS loopback client, directory lookups, directory reads, and file creation would all still need to be performed synchronously.

The second benefit of the SFS server as a loopback client is security. The SFS server is of course trusted, as it may be called upon to serve or modify any file. The server must therefore be careful not to perform any operation not permitted to the requesting users. Had the server been implemented on top of the normal file system interface, it would also have needed to perform access control—deciding on its own, for instance, whether or not to honor a request to delete a file. Making such decisions correctly without race conditions is actually quite tricky to do given only the Unix file system interface.¹

¹ In the example of deleting a file, the server would need to change its working directory to that of the file. Otherwise, between the server's access check and its *unlink* system call, a bad user could replace the di-

rectory with a symbolic link, thus tricking the server into deleting a file in a different (unchecked) directory. Cross-directory renames are even worse—they simply cannot be implemented both atomically and securely. An alternative approach might be for the server to drop privileges before each file system operation, but then unprivileged users could send signals to the server and kill it.

The final benefit of having used a loopback client is in avoiding limits on the number of open files. The total number of open files on SFS clients connected to an SFS server may exceed the maximum number of open files allowed on the server. As an NFS loopback client, the SFS server can access a file without needing a dedicated file descriptor.

A user of the SFS toolkit actually prototyped an SFS server that used the POSIX interface rather than act as a loopback client. Even without implementing leases on attributes, user authentication, or unique *st_ino* fields, the code was almost as large as the production SFS server and considerably more bug-prone. The POSIX server had to jump through a number of hoops to deal with such issues as the maximum number of open files.

5.1 Limitations of NFS loopback clients

The only major limitation on NFS loopback clients is that they must run as root. Unprivileged programs cannot access the file system with the NFS interface. A related concern is that the value of NFS file handles must be carefully guarded. If even a single file handle of a directory is disclosed to an untrusted user, the user can access any part of the file system as any user. Fortunately, the SFS RPC compiler provides a solution to this problem. One can easily traverse an arbitrary NFS data structure and encrypt or decrypt all file handles encountered. The SFS toolkit contains a support routine for doing so.

A final annoyance of loopback clients is that the file systems they access must be exported via NFS. The actual mechanics of exporting a file system vary significantly between versions of Unix. The toolkit does not yet have a way of exporting file systems automatically. Thus, users must manually edit system configuration files before the loopback client will run.

rectory with a symbolic link, thus tricking the server into deleting a file in a different (unchecked) directory. Cross-directory renames are even worse—they simply cannot be implemented both atomically and securely. An alternative approach might be for the server to drop privileges before each file system operation, but then unprivileged users could send signals to the server and kill it.

# Lines	Function
19	includes and global variables
22	command-line argument parsing
8	locate and spawn <i>nfsmounter</i>
5	get server's IP address
11	ask server's portmap for NFS port
14	ask server for NFS file handle
20	call <i>nfsmounter</i> for mount
20	relay NFS calls
119	Total

Figure 4: Lines of code in *dumbfs*

6 Applications of the toolkit

The SFS toolkit has been used to build a number of systems. SFS itself is a distributed file system consisting of two distinct protocols, a read-write and a read-only protocol. On the client side, each protocol is implemented by a separate loopback server. On the server side, the read-write protocol is implemented by a loopback client. (The read-only server uses the POSIX interface.) A number of non-SFS file systems have been built, too, including a file system interface to CVS, a file system to FTP/HTTP gateway, and file system interfaces to several databases.

The toolkit also lets one develop distributed file systems and fit them into the SFS framework. The SFS read-write protocol is very similar to NFS 3. With few modifications, therefore, one can transform a loopback server into a network server accessible from any SFS client. SFS's libraries automatically handle key management, encryption and integrity checking of session traffic, user authentication, and mapping of user credentials between local and remote machines. Thus, a distributed file system built with the toolkit can without much effort provide a high level of security against network attacks.

Finally, the asynchronous I/O library from the toolkit has been used to implement more than just file systems. People have used it to implement TCP proxies, caching web proxies, and TCP to UDP proxies for networks with high loss. Asynchronous I/O is an extremely efficient tool for implementing network servers. A previous version of the SFS toolkit was used to build a high performance asynchronous SMTP mail server that survived a distributed mail-bomb attack. (The “hybris worm” infected thousands of machines and made them all send mail to our server.)

6.1 *dumbfs* – A simple loopback server

To give a sense for the complexity of using the SFS toolkit, we built *dumbfs*, the simplest possible loopback

file system. *dumbfs* takes as arguments a server name, a pathname, and a mount point. It creates a loopback NFS mount on the mount point, and relays NFS calls to a remote NFS server. Though this functionality may sound worthless, such a utility does actually have a use. Because the SFS RPC libraries will trace and pretty-print RPC traffic, *dumbfs* can be used to analyze exactly how an NFS client and server are behaving.

The implementation of *dumbfs* required 119 lines of code (including blank lines for readability), as shown in Figure 4. Notably missing from the breakdown is any clean-up code. *dumbfs* does not need to catch any signals. If it dies for any reason, *nfsmounter* will clean up the mount point.

Figure 5 shows the implementation of the NFS call relaying code. Function *dispatch* is called for each NFS RPC. The first two lines manipulate the “auth unix parameters” of the NFS call—RPC terminology for user and group IDs. To reuse the same credentials in an outgoing RPC, they must be converted to an AUTH * type. The AUTH * type is defined by the RPC library in the operating system's C library, but the authopaque routines are part of the SFS toolkit.

The third line of *dispatch* makes an outgoing NFS RPC. *nfsc* is an RPC handle for the remote NFS server. *nc->getvoidarg* returns a pointer to the RPC argument structure, cast to void *. *nc->getvoidres* similarly returns a pointer to the appropriate result type, also cast to void *. Because the RPC library is asynchronous, *nfsc->call* will return before the RPC completes. *dispatch* must therefore create a callback, using *wrap* to bundle together the function *reply* with the argument *nc*.

When the RPC finishes, the library makes the callback, passing an additional argument of type *clnt_stat* to indicate any RPC-level errors (such as a timeouts). If such an error occurs, it is logged and propagated back as the generic RPC failure code *SYSTEM_ERR*. *warn* is the toolkit's asynchronous logging facility. The syntax is similar to C++'s *cout*, but *warn* additionally converts RPC and NFS enum error codes to descriptive strings. If there is no error, *reply* simply returns the result data structure just filled in by the outgoing NFS RPC.

6.1.1 *dumbfs* performance

To analyze the inherent overhead of a loopback server, we measured the performance of *dumbfs*. We ran experiments between two 800 MHz Pentium III machines, each with 256 MBytes of PC133 RAM and a Seagate ST318451LW disk. The two machines were connected with 100 Mbit/sec switched Ethernet. The client ran FreeBSD 4.2, and the server OpenBSD 2.8.

```

void
dispatch (nfscall *nc)
{
    static AUTH *ao = authopaque_create ();
    authopaque_set (ao, nc->getaup ());
    nfsc->call (nc->proc (), nc->getvoidarg (), nc->getvoidres (),
                 wrap (reply, nc), ao);
}

static void
reply (nfscall *nc, enum clnt_stat stat)
{
    if (stat) {
        warn << "NFS server: " << stat << "\n";
        nc->reject (SYSTEM_ERR);
    }
    else
        nc->reply (nc->getvoidres ());
}

```

Figure 5: *dumbfs* dispatch routine

To isolate the loopback server's worst characteristic, namely its latency, we measured the time to perform an operation that requires almost no work from the server—an unauthorized *fchown* system call. NFS required an average of 186 μ sec, *dumbfs* 320 μ sec. Fortunately, raw RPC latency is a minor component of the performance of most real applications. In particular, the time to process any RPC that requires a disk seek will dwarf *dumbfs*'s latency. As shown in Figure 7, the time to compile emacs 20.7 is only 4% slower on *dumbfs* than NFS. Furthermore, NFS version 4 has been specifically designed to tolerate high latency, using such features as batching of RPCs. In the future, latency should present even less of a problem for NFS 4 loopback servers.

To evaluate the impact of *dumbfs* on data movement, we measured the performance of sequentially reading a 100 MByte sparse file that was not in the client's buffer cache. Reads from a sparse file cause an NFS server to send blocks of zeros over the network, but do not require the server to access the disk. This represents the worst case scenario for *dumbfs*, because the cost of accessing the disk is the same for both file systems and would only serve to diminish the relative difference between NFS 3 and *dumbfs*. NFS 3 achieved a throughput of 11.2 MBytes per second (essentially saturating the network), while *dumbfs* achieved 10.3. To get a rough idea of CPU utilization, we used *top* command to examine system activity while streaming data from a 50 GByte sparse file through *dumbfs*. The idle time stayed between 30–35%.

6.2 *cryptfs* – An encrypting file system

As a more useful example, we built an encrypting file system in the spirit of CFS [3]. Starting from *dumbfs*, it took two evenings and an additional 600 lines of C++ to build *cryptfs*—a cryptographic file system with a very crude interface. *cryptfs* takes the same arguments as *dumbfs*, prompts for a password on startup, then encrypts all file names and data.

cryptfs was inconvenient to use because it could only be run as root and unmounting a file system required killing the daemon. However, it worked so well that we spent an additional week building *cryptfsd*—a daemon that allows multiple encrypted directories to be mounted at the request of non-root users. *cryptfsd* uses almost the same encrypting NFS translator as *cryptfs*, but is additionally secure for use on machines with untrusted non-root users, provides a convenient interface, and works with SFS as well as NFS,

6.2.1 Implementation

The final system is 1,920 lines of code, including *cryptfsd* itself, a utility *cmount* to mount encrypted directories, and a small helper program *pathinfo* invoked by *cryptfsd*. Figure 6.2 shows a breakdown of the lines of code. By comparison, CFS is over 6,000 lines (though of course it has different features from *cryptfs*). CFS's NFS request handing code (its analogue of *nfs.C*) is 2,400 lines.

cryptfsd encrypts file names and contents using Rijn-

# Lines	File	Function
223	cryptfs.h	Structure definitions, inline & template functions, global declarations
90	cryptfsd.C	<i>main</i> function, parse options, launch <i>nfsmounter</i>
343	findfs.C	Translate user-supplied pathname to NFS/SFS server address and file handle
641	nfs.C	NFS dispatch routine, encrypt/decrypt file names & contents
185	afs.C	NFS server for /cfs, under which encrypted directories are mounted
215	adm.C	<i>cryptfsadm</i> dispatch routine—receive user mount requests
26	cryptfsadm.x	<i>cryptfsadm</i> RPC protocol for requesting mounts from <i>cryptfsd</i>
63	cmount.C	Utility to mount an encrypted directory
134	pathinfo.c	Helper program—run from <i>findfs.C</i> to handle untrusted requests securely
1,920	Total	

Figure 6: Lines of code in *cryptfsd*. The last two source files are for stand-alone utilities.

dael, the AES encryption algorithm. File names are encrypted in CBC mode, first forwards then backwards to ensure that every byte of the encrypted name depends on every byte of the original name. Symbolic links are treated similarly, but have a 48-bit random prefix added so that two links to the same file will have different encryptions.

The encryption of file data does not depend on other parts of the file. To encrypt a 16-byte block of plaintext file data, *cryptfsd* first encrypts the block’s byte offset in the file and a per-file “initialization vector,” producing 16 bytes of random-looking data. It exclusive-ors this data with the plaintext block and encrypts again. Thus, any data blocks repeated within or across files will have different encryptions.

cryptfsd could have used a file’s inode number or a hash of its NFS file handle as the initialization vector. Unfortunately, such vectors would not survive a traditional backup and restore. Instead, *cryptfsd* stores the initialization vector at the beginning of the file. All file contents is then shifted forward 512 bytes. We used 512 bytes though 8 would have sufficed because applications may depend on the fact that modern disks write aligned 512-byte sectors atomically.

Many NFS servers use 8 KByte aligned buffers. Shifting a file’s contents can therefore hurt the performance of random, aligned 8 KByte writes to a large file; the server may end up issuing disk reads to produce complete 8 KByte buffers. Fortunately, reads are not necessary in the common case of appending to files. Of course, *cryptfs* could have stored the initialization vector elsewhere. CFS, for instance, stores initialization vectors outside of files in symbolic links. This technique incurs more synchronous disk writes for many metadata operations, however. It also weakens the semantics of the atomic rename operation. We particularly wished to avoid deviating from traditional crash-recovery semantics for operations like rename.

Like CFS, *cryptfs* does not handle sparse files properly. When a file’s size is increased with the *truncate* system call or with a *write* call beyond the file’s end, any unwritten portions of the file will contain garbage rather than 0-valued bytes.

The SFS toolkit simplified *cryptfs*’s implementation in several ways. Most importantly, every NFS 3 RPC (except for NULL) can optionally return file attributes containing file size information. Some operations additionally return a file’s old size before the RPC executed. *cryptfs* must adjust these sizes to hide the fact that it shifts file contents forward. (Also, because Rijndael encrypts blocks of 16 bytes, *cryptfs* adds 16 bytes of padding to files whose length is not a multiple of 16.)

Manually writing code to adjust file sizes wherever they appear in the return types of 21 different RPC calls would have been a daunting and error-prone task. However, the SFS toolkit uses the RPC compiler’s data structure traversal functionality to extract all attributes from any RPC data structure. Thus, *cryptfs* only needs a total of 15 lines of code to adjust file sizes in all RPC replies.

cryptfs’s implementation more generally benefited from the SFS toolkit’s single dispatch routine architecture. Traditional RPC libraries call a different service function for every RPC procedure defined in a protocol. The SFS toolkit, however, does not demultiplex RPC procedures. It passes them to a single function like *dispatch* in Figure 5. When calls do not need to be demultiplexed (as was the case with *dumbfs*), this vastly simplifies the implementation.

File name encryption in particular was simplified by the single dispatch routine architecture. A total of 9 NFS 3 RPCs contain file names in their argument types. However, for 7 of these RPCs, the file name and directory file handle are the first thing in the argument structure. These calls can be handled identically. *cryptfs* therefore implements file name encryption in a switch statement with 7 cascaded “case” statements and two special cases.

(Had the file names been less uniformly located in argument structures, of course, we could have used the RPC traversal mechanism to extract pointers to them.)

Grouping code by functionality rather than RPC procedure also results in a functioning file system at many more stages of development. That in turn facilitates incremental testing. To develop *cryptfs*, we started from *dumbfs* and first just fixed the file sizes and offsets. Then we special-cased read and write RPCs to encrypt and decrypt file contents. Once that worked, we added file name encryption. At each stage we had a working file system to test.

A function to “encrypt file names whatever the RPC procedure” is easy to test and debug when the rest of the file system works. With a traditional demultiplexing RPC library, however, the same functionality would have been broken across 9 functions. The natural approach in that case would have been to implement one NFS RPC at a time, rather than one feature at a time, thus arriving at a fully working system only at the end.

6.2.2 *cryptfs* performance

To evaluate *cryptfs*’s performance, we measured the time to untar the emacs 20.7 software distribution, configure the software, compile it, and then delete the build tree. Figure 7 shows the results. The white bars indicate the performance on FreeBSD’s local FFS file system. The solid gray bars show NFS 3 over UDP. The solid black bars show the performance of *cryptfs*. The leftward slanting black bars give the performance of *dumbfs* for comparison.

The untar phase stresses data movement and latency. The delete and configure phases mostly stress latency. The compile phase additionally requires CPU time—it consumes approximately 57 seconds of user-level CPU time as reported by the time command. In all phases, *cryptfs* is no more than 10% slower than NFS3. Interestingly enough, *cryptfs* actually outperforms NFS 3 in the delete phase. This does not mean that *cryptfs* is faster at deleting the same files. When we used NFS 3 to delete an emacs build tree produced by *cryptfs*, we observed the same performance as when deleting it with *cryptfs*. Some artifact of *cryptfs*’s file system usage—perhaps the fact that almost all file names are the same length—results in directory trees that are faster to delete.

For comparison, we also ran the benchmark on CFS using the Blowfish cipher.² The black diagonal stripes

²Actually, CFS did not execute the full benchmark properly—some directories could not be removed, putting rm into an infinite loop. Thus, for CFS only, we took out the benchmark’s rm -rf command, instead deleting as much of the build tree as possible with find/xargs and then renaming the emacs-20.7 directory to a garbage name.

labeled “CFS-async” show its performance. CFS outperforms even straight NFS 3 on the untar phase. It does so by performing asynchronous file writes when it should perform synchronous ones. In particular, the *fsync* system call does nothing on CFS. This is extremely dangerous—particularly since CFS runs a small risk of deadlocking the machine and requiring a reboot. Users can lose the contents of files they edit.

We fixed CFS’s dangerous asynchronous writes by changing it to open all files with the O_FSYNC flag. The change did not affect the number system calls made, but ensured that writes were synchronous, as required by the NFS 2 protocol CFS implements. The results are shown with gray diagonal stripes, labeled “CFS-sync.” CFS-sync performs worse than *cryptfs* on all phases. For completeness, we also verified that *cryptfs* can beat NFS 3’s performance by changing all writes to unstable and giving fake replies to COMMIT RPCs—effectively what CFS does.

Because of the many architectural differences between *cryptfs* and CFS, the performance measurements should not be taken as a head-to-head comparison of SFS’s asynchronous RPC library with the standard *libc* RPC facilities that CFS uses. However, CFS is a useful package with performance many that people find acceptable given its functionality. These experiments show that, armed with the SFS toolkit, the author could put together a roughly equivalent file system in just a week and a half.

Building *cryptfsd* was a pleasant experience, because every little piece of functionality added could immediately be tested. The code that actually manipulates NFS RPCs is less than 700 lines. It is structured as a bunch of small manipulations to NFS data structures being passed between and NFS client and server. There is a mostly one-to-one mapping from RPCs received to those made. Thus, it is easy for *cryptfs* to provide traditional crash-recovery semantics.

7 Summary

User-level software stands to gain much by using NFS over the loopback interface. By emulating NFS servers, portable user-level programs can implement new file systems. Such loopback servers must navigate several tricky issues, including deadlock, vnode and mount point locking, and the fact that a loopback server crash can wedge a machine and require a reboot. The SFS file system development toolkit makes it relatively easy to avoid these problems and build production-quality loopback NFS servers.

The SFS toolkit also includes an asynchronous RPC library that lets applications access the local file system using NFS. For aggressive applications, NFS can actu-

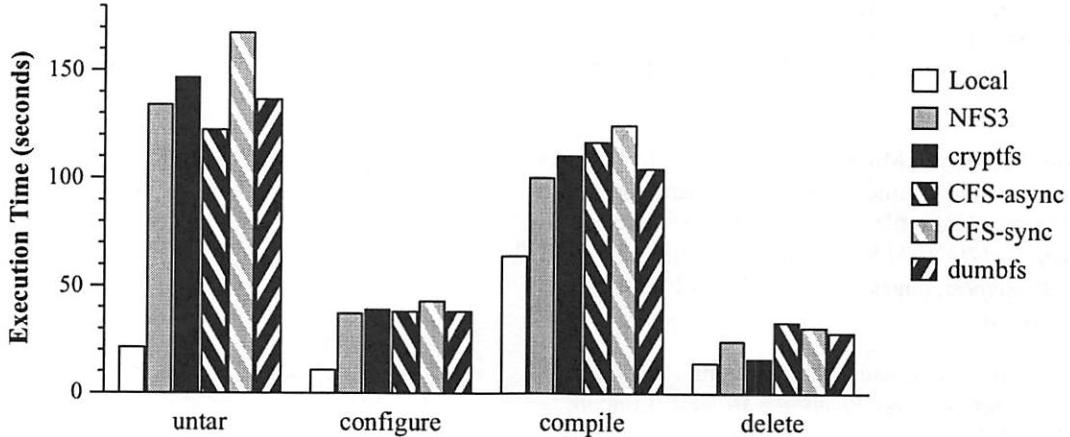


Figure 7: Execution time of emacs untar, configure, compile, and delete.

ally be a better interface than the traditional *open/close/read/write*. NFS allows asynchronous access to the file system, even for operations such as file creation that are not asynchronously possible through system calls. Moreover, NFS provides a lower-level interface that can help avoid certain race conditions in privileged software.

Acknowledgments

The author is grateful to Frans Kaashoek for implementing the POSIX-based SFS server mentioned in Section 5, for generally using and promoting the SFS toolkit, and for his detailed comments on several drafts of this paper. The author also thanks Marc Waldman, Benjie Chen, Emmett Witchel, and the anonymous reviewers for their helpful feedback. Finally, thanks to Yoonho Park for his help as shepherd for the paper.

Availability

The SFS file system development toolkit is free software, released under version 2 of the GNU General Public License. The toolkit comes bundled with SFS, available from <http://www.fs.net/>.

References

- [1] Albert D. Alexandrov, Maximilian Ibel, Klaus E. Schauer, and Chris J. Scheiman. Extending the operating system at the user level: the Ufo global file system. In *Proceedings of the 1997 USENIX*, pages 77–90. USENIX, January 1997.
- [2] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.
- [3] Matt Blaze. A cryptographic file system for unix. In *1st ACM Conference on Communications and Computing Security*, pages 9–16, November 1993.
- [4] B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, Network Working Group, June 1995.
- [5] Brent Callaghan and Tom Lyon. The automounter. In *Proceedings of the Winter 1989 USENIX*, pages 43–51. USENIX, 1989.
- [6] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, February 1999.
- [7] Vincent Cate. Alex—a global filesystem. In *Proceedings of the USENIX File System Workshop*, May 1992.
- [8] Jeremy Fitzhardinge. UserFS. <http://www.goop.org/~jeremy/userfs/>.
- [9] David K. Gifford, Pierre Jouvelot, Mark Sheldon, and James O’Toole. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 16–25, Pacific Grove, CA, October 1991. ACM.
- [10] Cary G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*. ACM, 1989.
- [11] John S. Heidemann and Gerald J. Popek. File system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, February 1994.

- [12] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.
- [13] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 124–139, Kiawa Island, SC, 1999. ACM.
- [14] Jan-Simon Pendry and Nick Williams. *Amd The 4.4 BSD Automounter Reference Manual*. London, SW7 2BZ, UK. Manual comes with amd software distribution.
- [15] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX*, pages 119–130, Portland, OR, 1985. USENIX.
- [16] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS version 4 protocol. RFC 3010, Network Working Group, December 2000.
- [17] Assar Westerlund and Johan Danielsson. Arla—a free AFS client. In *Proceedings of the 1998 USENIX, Freenix track*, New Orleans, LA, June 1998. USENIX.
- [18] Erez Zadok and Alexander Dupuy. HLFSD: Delivering email to your \$HOME. In *Proceedings of the Systems Administration Conference (LISA VII)*, Monterey, CA, November 1993. USENIX.
- [19] Erez Zadok and Jason Nieh. FiST: A language for stackable file systems. In *Proceedings of the 2000 USENIX*. USENIX, June 2000.

Charm: An I/O-Driven Execution Strategy for High-Performance Transaction Processing

Lan Huang

*Department of Computer Science
State University of New York
Stony Brook, NY 11790
lanhuang@cs.sunysb.edu*

Tzi-cker Chiueh

*Department of Computer Science
State University of New York
Stony Brook, NY 11790
chiueh@cs.sunysb.edu*

Abstract

The performance of a transaction processing system whose database is not completely memory-resident critically depends on the amount of physical disk I/O required. This paper describes a high-performance transaction processing system called *Charm*, which aims to reduce the concurrency control overhead by minimizing the performance impacts of disk I/O on lock contention delay. In existing transaction processing systems, a transaction blocked by lock contention is forced to wait while the transaction currently holding the contended lock performs physical disk I/O. A substantial portion of a transaction's lock contention delay is thus attributed to disk I/Os performed by other transactions. *Charm* implements a two-stage transaction execution (TSTE) strategy, which makes sure that all the data pages that a transaction needs are memory-resident before it is allowed to lock database pages. Moreover, *Charm* supports an optimistic version of the TSTE strategy (OTSTE), which further eliminates unnecessary performance overhead associated with TSTE. Another TSTE variant (HTSTE) attempts to achieve the best of both TSTE and OTSTE by executing only selective transactions using TSTE and others using OTSTE. *Charm* has been implemented on the Berkeley DB package and requires only a trivial modification to existing applications. Performance measurements from a fully operational *Charm* prototype based on the TPC-C workload demonstrate that *Charm* out-performs conventional transaction processing systems by up to 164% in transaction throughput, when the application's performance is limited by lock contention.

1 Introduction

High-performance transaction processing systems have seen a resurgent interest within the explosively growing E-commerce community, especially after the publicly reported "melt-down" of the on-line electronic trading system of several well-established stock brokerage houses. The user-perceived response time of a transaction consists of three components: CPU processing time, disk I/O time, and waiting time due to lock contention. For high-throughput transaction processing systems used in banking and stock trading applications, CPU time is typically insignificant compared to disk I/O time. Unless the underlying database is fully memory-resident, read disk I/O cannot be completely eliminated. Moreover, database logging and data persistence require write disk I/O even with main-memory database management systems. Most on-line transaction processing systems, although equipped with a large amount of physical memory to reduce the number of disk I/Os, do not necessarily have the luxury to keep the entire database memory-resident. Lock contention delay is itself often dependent on the amount of disk I/O, because existing transaction systems allow transactions that are holding locks to perform disk I/O and thus lengthen the waiting time of those transactions that are blocked by the held locks.

While extensive research has been done in the concurrency control area to improve the throughput of transaction processing systems, this paper presents the design, implementation and evaluation of a transaction execution strategy that is *orthogonal* and thus *complementary* to existing concurrency control algorithms. By re-arranging the order of execution of I/O operations within each individual transaction, the Two-Stage Transaction Execution

(TSTE) strategy described in this paper greatly reduces the average response time of individual transactions and improves the overall throughput of the transaction processing system. The authors wish to emphasize that the TSTE strategy is *not* yet another concurrency control algorithm, because its goal is not to reduce the number of lock conflicts, but to reduce the contention delay associated with each lock conflict.

Specifically, the goal of TSTE is to reduce the lock contention delay in disk-resident transaction processing systems to the same level as that experienced by memory-resident transaction processing systems. TSTE runs each transaction in two stages, a *fetch* stage and an *operate* stage. In the *fetch* stage, TSTE brings all the data pages required by a transaction to main memory and pins them down. *No locks are acquired in this stage.* In the *operate* stage, TSTE actually executes the transaction in exactly the same way as in traditional transaction processing systems, i.e., acquiring/releasing and waiting for locks, etc. Because the data pages required by a transaction are guaranteed to be memory-resident in the *operate* stage, the contention among transactions running in this stage is identical to the case when these transactions are running against a memory-resident database. In other words, as far as lock contention is concerned, TSTE turns a disk-based transaction processing system into a memory-resident transaction processing system by decoupling disk I/O from lock acquisition/release: in the *fetch* stage, a TSTE transaction performs disk I/O but not locking, and vice versa in the *operate* stage. Consequently the following invariant always holds true for an ideal decoupled transaction processing system: *The lock contention delay that a transaction experiences never includes the disk I/O time of another transaction.* Due to practical implementation issues and inter-transaction data sharing behavior, the above invariant may not always hold in TSTE. Therefore, despite its ability to reduce the lock contention delay, TSTE also incurs additional performance overhead. We have developed several optimization techniques to effectively reduce this extra performance cost.

We have implemented the first *Charm* prototype, including several performance optimizations, based on the Linux-based Berkeley DB package, version 2.4.14 [1, 2, 17] and carried out a detailed performance study on this prototype using a standard online transaction processing benchmark, TPC-C. The rest of this paper is organized as follows. Section 2

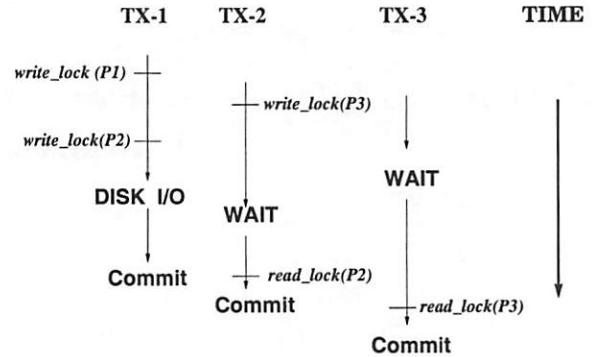


Figure 1: An example scenario in which transactions that are blocked due to lock contention (TX-2 and TX-3) actually experience the delay of the physical disk I/O performed by another transaction (TX-1).

reviews previous related research efforts on the reduction of concurrency control cost. Section 3 describes in more detail the TSTE strategy, as well as its optimized variants. Section 4 presents the results and analysis of a detailed performance study on the *Charm* prototype. Section 5 concludes this paper with a summary of main results from this research and an outline of on-going work.

2 Related Work

The idea of TSTE was originally proposed in earlier work [14] and has been analyzed by Franaszek et. al. [12, 13], which exploited the concept of *access invariance* to perform speculative disk I/O, and compared TSTE with various combinations of optimistic concurrency control and two-phase locking (2PL) through a simulation-based study. None of these works included the optimistic TSTE and hybrid TSTE schemes discussed in this paper. Moreover, *Charm* is the first known implementation of TSTE in industrial-strength transaction processing software, and this paper is the first to report empirical performance measurements of TSTE from a system researcher's view.

TSTE is *not* a new concurrency control algorithm. While newer concurrency control algorithms exploit semantics of transactions [3] or objects of abstract data types [4, 5] to release locks as early as possible, TSTE takes a completely different approach by removing disk I/O time from lock contention delay. There is a superficial similarity between optimistic concurrency control algorithms [6] and TSTE, es-

pecially its optimistic variant, because they both adopt rollback as a mechanism to undo the effects of writes when the speculated assumptions do not hold. However, the underlying conditions of speculation, as well as when to rollback, are very different for these two algorithms. Optimistic concurrency control assumes that lock contention is rare, and therefore proceeds with data accesses without acquiring locks. It is required to roll back the transaction when lock conflicts are detected at the *end* of the transaction. TSTE, on the other hand, assumes that disk I/O is rare, and undoes all the intermediate writes when it detects that a transaction is going to perform the *first* physical disk I/O. Whereas optimistic concurrency control attempts to reduce the lock acquisition/release overhead, TSTE aims to reduce the portion of lock contention delay attributed to disk I/O.

The idea of *decoupled architecture* [7] in computer architecture literature partially motivates the TSTE strategy. It was originally proposed to bridge the speed gap between CPU execution and memory access, but has been extended to address the performance difference between memory and disk [8]. The main idea is to statically split a program into a computation part and a data access part, and to run the data access part ahead of the computation part so that the data required by the computation part has been brought into cache (memory) from memory (disk) by the time it is actually needed. TSTE's *fetch* stage is essentially a (close to) perfect prefetching mechanism. However, the performance advantage of disk data prefetching, lies not in decreased data access delays, but in the reduction of lock contention delays as seen by other transactions that contend for locks that are being held. Recently Chang [15] developed an automatic binary modification tool that modifies existing binaries to perform speculative execution in order to generate prefetching hints.

TSTE is different from other file system or database prefetching research [9, 10] because its goal is not really about cutting down the number of physical disk I/Os, but to decouple disk I/Os from lock acquisitions. As the database size increases and application access patterns become more complicated, the effectiveness of file system/database prefetching methods decreases, but the usefulness of TSTE increases because it is more likely for a transaction to encounter disk I/O while holding locks.

3 Two-Stage Transaction Execution

3.1 Basic Algorithm

The fundamental observation motivating the Two-Stage Transaction Execution (TSTE) strategy is that the average lock contention delay in main memory database systems is much smaller than disk-resident database systems because of the absence of disk I/O. For example, in Figure 1 Transaction 2 is blocked because of the read access to P2, which is write-locked by Transaction 1, and Transaction 3 is blocked because of the read access to P3, which is write-locked by Transaction 2. As a result, both Transaction 2 and 3 experience the delay associated with the disk I/O performed by Transaction 1. In general, the delay of a physical disk I/O could appear as part of the response time of one or multiple transactions. The ultimate goal of TSTE is to ensure that each disk I/O's access delay contributes to the response time of exactly one transaction, the one that initiates the disk I/O.

Table 1 shows the average percentage of a transaction's lock contention delay that is due to disk I/Os performed by other transactions, measured on the Berkeley DB package running the TPC-C workload with the *Warehouse*¹ parameter set to five, which represents a database larger than 1 GByte. The testing machine has an 800 MHz PIII CPU, 512 Mbytes of memory is used for the user level buffer cache. There are a total of 640 Mbytes physical memory in this machine. The database tables reside on three 5400-RPM disks: one disk holds transaction logs; the other two disks hold all other data. This table is meant to illustrate the extent of the potential performance improvement if disk I/O is completely decoupled from lock contention. For example, up to 93.3% of the lock contention delay could be eliminated when the number of concurrent transactions is two. As the number of concurrent transactions increases, this percentage decreases because delay due to true data contention starts to dominate.

The basic idea of TSTE is to split the execution of each transaction into two stages. In the *fetch* stage, TSTE performs all the necessary disk I/Os to bring the data pages that a transaction needs into main memory and pins them down, by executing the transaction once *without updating the database*.

¹ Warehouse is the database size scaling factor.

One possible implementation of the *fetch* stage is to keep a local copy of the updates without committing them to the database at the end of the transaction. Another implementation, which is used in our system, is to skip all update operations and only bring into main memory necessary pages. In the second *operate* stage, the transaction is executed again, in the same way as in conventional transaction processing architectures. Because the *fetch* stage brings all required data pages into memory, transactions in the *operate* stage should never need to access the disks as long as access invariance holds. In those cases that the access invariance property does not hold, the transactions need to perform disk I/Os in the *operate* stage.

No. of Concurrent Transactions	Percentage of Lock Contention Delay due to Disk I/O
2	93.3%
4	80.9%
6	54.1%
8	34.8%
10	18.4%

Table 1: Average percentage of lock contention delay that a transaction experiences that is due to disk I/Os performed by other contending transactions, versus the number of concurrent transactions. The measurements are collected on the Berkeley DB package running the TPC-C workload with the *Warehouse* parameter set to five. The user-level database cache size is set to 512 Mbytes and CPU is an 800 MHz Pentium III.

The execution of a transaction in the *fetch* stage is special in two aspects. First, transactions executing in this stage do not lock database items before accessing them. Because transactions cannot hold locks on database items, it is impossible for one transaction in the *fetch* stage to wait for a lock held by another transaction. On the other hand, transactions in the *operate* stage never need to access disks, because the pages they need are brought into memory in the *fetch* stage. As a result, a transaction blocked on a lock should never experience, during the waiting period, any delay associated with disk I/O performed by the transaction currently holding the lock. With the TSTE strategy, the lock contention delay contains only CPU execution and queuing times and is thus much smaller than that in conventional transaction processing architectures.

Second, because transactions do not acquire locks

before accessing data, they should not modify data in the *fetch* stage, either. That is, transactions perform only read disk I/Os in the *fetch* stage, including those data pages that are to be written as well as the address generation computation for data page accesses, but skip all write operations. This guarantees that the result of executing transactions using TSTE is identical to that in conventional transaction processing architectures.

By separating disk I/O and lock acquisition into two mutually exclusive stages, TSTE eliminates the possibility of long lock contention delay due to disk I/O. However, TSTE itself incurs additional performance overhead. The fact that TSTE executes each transaction twice means that TSTE is doing redundant work compared to conventional transaction processing systems. For transactions whose CPU time is large, this redundant work may overshadow the performance gain from TSTE. Fortunately, the CPU processing time is typically small compared to disk I/O delay for applications that require high transaction throughput such as stock trading applications. However, for long running transactions, TSTE may require too much memory to hold fetched pages and may exceed memory resources.

Because TSTE does not acquire locks, it is possible that the data pages chosen to be brought into memory in the *fetch* stage are not the same as those needed in the *operate* stage. The reason is that between the *fetch* and *operate* stages, the pages that a transaction needs may change. In this case, some of the read disk I/Os performed in the *fetch* stage are useless and thus redundant. These disk I/Os correspond to *mis-prefetching*. For example, each *delivery* transaction in TPC-C is supposed to fulfill the oldest order in the database, and therefore should only start after the previous *delivery* transaction is finished. Prefetching the data record for the current “oldest order” before the previous *delivery* transaction is completed almost certainly lead to mis-prefetching. Furthermore, transactions that mis-prefetch actually need to perform read disk I/O in the *operate* stage to retrieve those pages that the *fetch* stage should have brought in. Therefore, the invariant that no transaction needs to wait for another transaction that is performing physical disk I/O does not hold for mis-prefetching transactions.

The scenario described in Figure 1 is not limited to lock contention incurred by false-sharing when only page-level or table-level locking is supported. Even for a system with fine-grain locking, lock contention

blocked by disk I/O can be reduced to a minimum using TSTE.

There is an exception to this two-stage execution strategy: accesses to database pages that are typically memory resident and for which the lock holding period is usually short. An example is accesses to highly concurrent data structures such as B-trees [11]. Specifically, in the current *Charm* prototype, accesses to the *intermediate* but not *leaf* nodes of B-trees are preceded by lock acquisitions even in the *fetch* stage. The separate treatment of B-tree's intermediate and leaf nodes prevents de-referencing of outdated and potentially dangling pointers, and reduces the probability of *mis-prefetching*, at the expense of a relatively minor performance cost.

3.2 Optimistic TSTE

To reduce TSTE's redundant CPU computation overhead, we developed an optimistic version of the TSTE (OTSTE) algorithm. With database buffering, not all transactions need to perform physical disk I/O in the *fetch* stage. Therefore, what a TSTE transaction attempts to achieve in the *fetch* stage, i.e., making sure that data pages needed in the *operate* stage are in main memory, may be redundant in some cases. Instead of always running a transaction in two stages, OTSTE starts each transaction in the *operate* stage. If all of a TSTE transaction's data page accesses hit in the database cache, the transaction completes successfully in one stage. If, however, the transaction indeed needs to access the disk, it "converts" itself to the *fetch* stage when the first such instance arises and subsequently proceeds to the *operate* stage as in standard TSTE.

The transition from the *operate* stage to the *fetch* stage in OTSTE involves the following three steps. First, the effects of all the database writes before this data access are un-done. Second, all the locks acquired in the *operate* stage are released. Third, all the data pages that this transaction has touched are pinned down in memory, in preparation for the subsequent *operate* stage.

Compared to TSTE, OTSTE eliminates the *fetch* stage altogether when the data pages required by a transaction are already memory-resident. However, OTSTE also incurs an additional overhead due to the transition from the *operate* stage to the *fetch* stage when a transaction needs to perform physi-

cal disk I/O and acquire locks. The major component of this stage-transition overhead is attributed to the undo of earlier database writes. As we will show later, OTSTE's performance gain resulting from eliminating unnecessary *fetch* stages does not out-weigh the overhead associated with rolling back updates.

3.3 Hybrid TSTE

Transactions running under TSTE incur redundant computation overhead when the transactions do not require physical disk I/O. OTSTE is meant to address this problem. However, in OTSTE, mis-prefetching and rollback overhead still exist. To further reduce these overheads, we developed another variant of TSTE called Hybrid TSTE (HTSTE), which classifies transactions running under TSTE into three types:

1. Transactions that encounter significant mis-prefetches in the *operate* stage;
2. Transactions that do not encounter mis-prefetches in the *operate* stage, and
3. Transactions that do not require any disk I/O.

HTSTE runs the first type of transactions as in traditional transaction processing systems, i.e., start them in the *operate* stage and do not convert them into the *fetch* stage even when physical disk I/O is needed. For this type of transaction, HTSTE does not incur rollback overhead. HTSTE runs the second type of transaction directly in the *fetch* stage, i.e., as in generic TSTE. HTSTE runs the third type of transactions in the *operate* stage but converts them into the *fetch* stage when physical disk I/O is needed, as in OTSTE.

The unique feature of HTSTE is that it does not require a priori knowledge of the transactions' data access behaviors. Instead, the decision of whether a transaction should be executed using TSTE, OTSTE, or a conventional transaction-processing model is made dynamically. By default, HTSTE executes a given transaction type using TSTE, and collects two accumulative statistics for all its instances. The first statistic, called *mis-prefetch ratio*, is the percentage of data accesses that lead to physical disk I/O in the *operate* stage, and

the other, called *conversion ratio*, is the percentage of execution instances of this transaction type that require two stages to complete. If a transaction type's *mis-prefetch ratio* is above a certain threshold, all its future instances will be executed as conventional transactions (Type 1). Otherwise if the transaction type's *conversion ratio* is above a certain threshold, all its future instances will continue to be executed under TSTE (Type 2). Otherwise, all the future instances of this transaction type will be executed using OTSTE (Type 3). We use a conservative value as threshold, e.g., only when more than 95% of one type of transaction requires no disk I/O, this transaction type will be started always in OTSTE. For the TPC-C benchmark², this prediction works out well: `neword` and `delivery` always require disk I/O during execution; `slev`, `ostat`, `payment` seldom do. If a certain transaction type does not display a clear trend in I/O behavior, it will be started as TSTE.

3.4 Cost Analyses

Let P ($P \leq 1$) be the percentage of transactions that need to perform physical disk I/O during their lifetime, and let's assume an ideal HTSTE implementation (i.e., All single-phased transactions do not need to perform disk I/O and all two-phased transaction do need to perform disk I/O.), the time to finish one transaction for each TSTE variant is as follows:

$$\text{TSTE: } \text{Transaction Time} = T_{fetch} + T_{operate} \quad (1)$$

$$\text{OTSTE: } \text{Transaction Time} = P * (T_{rollback} + T_{fetch} + T_{operate}) + (1 - P) * T_{operate} \quad (2)$$

$$\text{HTSTE: } \text{Transaction Time} = P * (T_{fetch} + T_{operate}) + (1 - P) * T_{operate} \quad (3)$$

T_{fetch} is the time spent in the *fetch* stage and $T_{operate}$ is that spent in the *operate* stage. $T_{rollback}$ is the time to undo operations of the previous transaction. Transactions running under TSTE experience both the *fetch* and the *operate* stage. Those running under OTSTE complete within a single stage with

²TPC-C contains five types of transactions: `neword`, `payment`, `slev`, `ostat`, and `delivery`. `Neword` places a new order for a customer. `Payment` clears the balance of a customer. `Slev` checks the stock level. `Ostat` checks the order status. `Delivery` delivers an order.

a probability of $(1 - P)$ and need to spend an additional rollback overhead with a probability of P . HTSTE categorizes transactions into I/O transactions and non-I/O transactions. Those transactions that need to perform disk I/O are executed in two stages, while those non-I/O transactions start with the *operate* stage directly without going through the *fetch* stage. HTSTE performs the best since it does not incur unnecessary rollback or fetch overhead. The performance difference among different TSTE variants depends on P , the rollback overhead and the relative costs of the *fetch* and *operate* stage. Note that here we are assuming an ideal implementation of HTSTE, which can correctly determine whether a transaction needs to perform disk I/O at run time. This may not be always the case in practice.

3.5 Prototype Implementation

We have implemented the TSTE prototype on the Berkeley DB package, version 2.4.14 [1], which is an industrial-strength transaction processing software library that has been used in several highly visible web-related companies such as Netscape. Berkeley DB supports page-level locking as well as error recovery through write-ahead logging. Berkeley DB supports an asynchronous database logging mode where commit records are not forced to disk, but are written lazily as the in-memory log buffer fills.

The fundamental difference between TSTE and traditional transaction processing systems lies in the *fetch* stage. What the TSTE prototype needs to do is to execute an input transaction twice, to *skip* all the lock acquisition requests and updates in the *fetch* stage and to pin down the database cache pages that are brought into memory in the first stage. When transactions are executed the second time, they are processed the same way as in Berkeley DB. HTSTE dynamically collects internal statistics and decides which type of transaction to start with the *fetch* stage and which to start with the *operate* stage.

To implement OTSTE, we modified Berkeley DB so that it starts each transaction in the *operate* stage, and pins down all the data pages accessed. When a TSTE transaction encounters a database cache miss for the first time, the system converts the transaction from the *operate* stage to the *fetch* stage. During this transition, all the effects of writes are un-

done, i.e., the database state is rolled back to the beginning of the transaction, and all the locks acquired by this transaction so far are released. Finally, the transaction re-starts by re-executing the missed database cache access, but this time in the *fetch* stage.

Charm requires minor modifications to the programming interface that Berkeley DB provides to its applications to keep track of I/O behavior of each type of transaction online. Although TSTE executes each transaction twice, the operations in the transactions do not have to be idempotent because the execution in the first stage never results in persistent side effects.

4 Performance Evaluation

4.1 Experiment Setup

To evaluate the performance of TSTE, we ran the TSTE prototype on an 800-MHz Pentium-III machine with 640 Mbytes of physical memory running the Linux kernel version 2.2.5 and compared its performance measurements against those from Berkeley DB, which uses a traditional transaction processing architecture. We used the standard transaction processing benchmark TPC-C. This study focuses only on the throughput of **neword** transactions. In the workload mix, **neword** accounts for 43%, **payment** 43%, and **ostat**, **slev** and **delivery** each 4.7%. *Warehouse* is the parameter to scale the database size. The *Warehouse* parameter (*w*) in TPC-C was set to five, corresponding to a total database size of over 1.0 GByte, which grows during each test run as new records are inserted. The database cache size used in this study was 512-Mbytes unless otherwise stated.

Each data point is the average result of ten runs, each of which consists of more than 10,000 transactions, which is sufficiently long for the system to reach a steady state. Before starting each run, we ran a number of transactions against the database to warm up the database cache. Unless otherwise stated, the number of warm-up transactions is 80,000. Each transaction is executed as a separate user process, and the number of concurrent transactions remains fixed throughout each run. All the following reported measurements are performed

at the user level. All the runs assume page-level locking. We set the log buffer size to 500 Kbytes and turned on the asynchronous logging mechanism, which eliminated most disk I/Os associated with transaction commits in our tests. This set-up ensures that the performance bottleneck lies mainly in lock waiting and data accessing disk I/O rather than logging disk I/O. All tests were run on a single computer and thus the reported performance measurements did not include network access delays, as in standard client-server set-ups. In the experiments, we compared Berkeley DB, TSTE OTSTE and HTSTE by varying the following workload/system parameters: the number of concurrent transactions, the database cache size and thus database cache hit ratio, and the computation time.

Three 5400-RPM IDE disks are used in the experiments. The log resides on disk 1, the **orderline** table resides on disk 2 and all other tables reside on disk 3. The write cache of the log disk is turned off to protect data integrity. **Neword** throughput is reported as system throughput as the TPC-C benchmark requires, whose metric unit is tpmC (transaction per minute).

4.2 Overall Comparison

We use the *transaction throughput* as our performance metrics, which is the ratio between the total number of **neword** transactions that succeed and the elapsed time required to complete a run of test transactions. Note that some transactions in the run are aborted to resolve deadlocks. The un-modified Berkeley DB package serves as the baseline case that corresponds to standard transaction processing systems based on two-phase locking. In HTSTE, **neword** and **delivery** transaction start in the *fetch* stage; **payment**, **ostat** and **slev** start in the *operate* stage.

Figure 2 shows the throughput comparison and Figure 3 shows the lock contention comparison between HTSTE and Berkeley DB when a different number of concurrent transactions are running simultaneously. In this case, the database cache is 512 Mbytes, which gives a database cache hit ratio of 99.7%. HTSTE out-performs Berkeley DB by up to 164% in transaction throughput (when concurrency is four). HTSTE is better than Berkeley DB except in the degenerate case when the number of concurrent transactions is one, where there is no

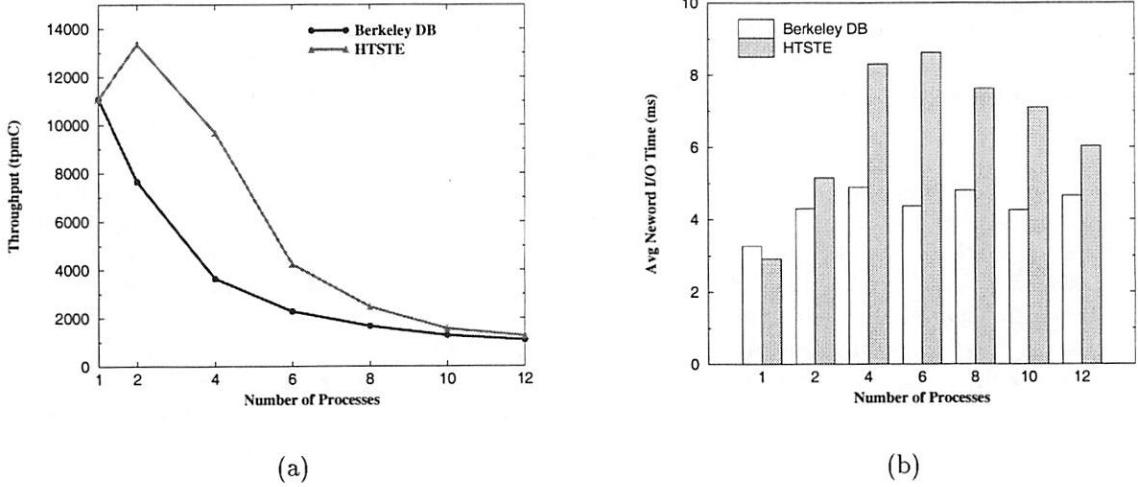


Figure 2: Overall system throughput (a) and average I/O time (b) per `neword` transaction comparison between HTSTE and Berkeley DB with different concurrency levels.

Concurrency	1	2	4	6	8	10	12
Berkeley DB	13.90	17.75	17.85	14.76	13.45	12.21	11.50
HTSTE	13.35	20.74	32.22	26.30	19.63	16.86	13.55

Table 2: Average file system read/write time comparison with different concurrency levels. The values are in terms of msec.

lock contention delay for HTSTE to reduce in the first place. And as concurrency level reaches 10–12, HTSTE and Berkeley DB have minor performance difference. The performance difference between HTSTE and Berkeley DB increases initially with the number of concurrent transactions until the concurrency reaches four. Before this point, lock contention delay dominates the transaction response time and therefore the reduction in lock contention delay that HTSTE affords plays an increasingly important role as lock contention delay increases with concurrency. After this point, the performance difference between HTSTE and Berkeley DB starts to decrease with the number of concurrent transactions because the reduction in lock contention delay becomes less significant percentage-wise with respect to the transaction response time.

To understand why HTSTE out-performs Berkeley DB, let's examine the three components of a transaction's response time: computation, disk I/O, and lock contention. The average computation time of a transaction under HTSTE is only slightly longer than that of Berkeley DB because of TSTE's two-pass execution. Figure 2(b) and Figure 3 show the comparison between HTSTE and Berkeley DB in

the average I/O time and lock contention time per `neword` transaction. As expected, HTSTE significantly reduces the lock contention delay, by up to a factor of 10 when compared to Berkeley DB (Figure 3(a) 2 processes). However, HTSTE still experiences noticeable lock contention delays, because mis-prefetching in the *fetch* stage leads to physical disk I/O in the *operate* stage, which in turn lengthens the lock contention time. As transaction concurrency increases, the probability of mis-prefetching grows, and consequently the difference in lock contention delay between HTSTE and Berkeley DB decreases.

Surprisingly, HTSTE fares worse than Berkeley DB in the average disk I/O time when more than two concurrent transactions are running simultaneously (Figure 2(b)). The total number of disk I/Os issued in both cases is roughly the same and the longer average I/O time in HTSTE is due to disk queuing. Table 2 shows that average file system read/write time for HTSTE is longer than Berkeley DB. We consider that the file system read/write time closely reflects the disk I/O latency without file system cache effects. The reason is that in our experiments, the file system cache is minimized

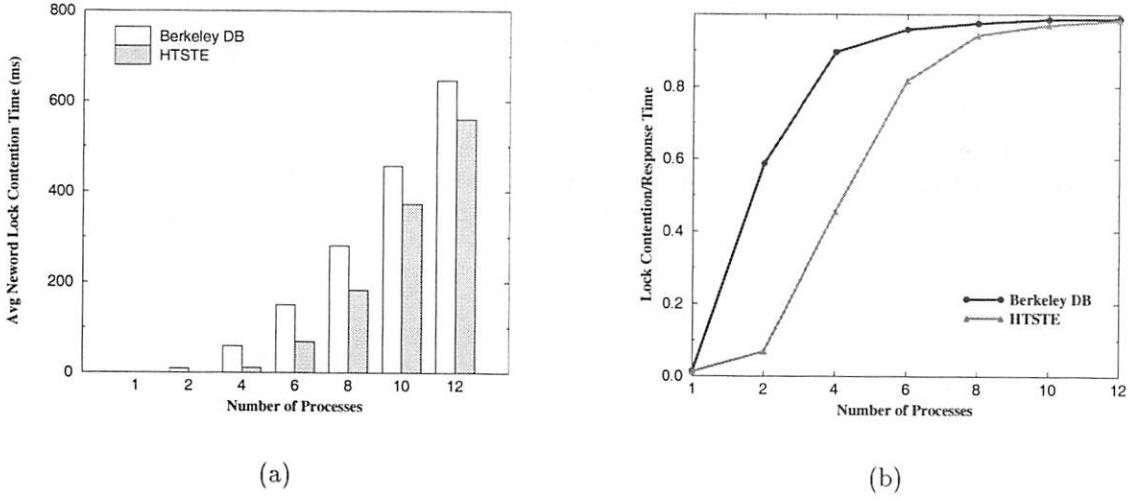


Figure 3: Average lock contention time per `neword` transaction (a) and the ratio between average lock contention time and average response time (b).

Concurrency	1	2	4	6	8	10	12
Berkeley DB	0.0%	0.7%	1.8%	2.8%	3.7%	4.6%	5.4%
HTSTE	0.0%	0.1%	0.4%	1.6%	3.2%	4.1%	5.0%

Table 3: The percentage of aborted transactions under different concurrency levels.

to almost zero bytes and the Berkeley DB package maintains its own user-level buffer cache. Because HTSTE allows transactions to perform their disk I/Os as soon as possible, it is more likely that disk I/Os are clustered and thus experience longer disk queuing delay. In contrast, under Berkeley DB the disk I/Os are more spread out because transactions are interlocked by lock contention. Therefore the TSTE strategy presents an interesting design trade-off between decreasing lock contention delay and increasing disk I/O time. In general, it is easier to invest more hardware to address the problem of longer disk queuing delay, e.g., by adding more disks, than to lowering the lock contention delay. Therefore, TSTE represents an effective approach to build more scalable transaction processing systems by taking advantage of increased hardware resources.

A minor benefit of HTSTE's reduced lock contention delay is the decreasing number of deadlocks (Table 3). This allows HTSTE to complete more transactions successfully within a given period of time than Berkeley DB, although the contribution of this is less than 2% of the throughput difference between HTSTE and Berkeley DB.

When $w = 1$, the initial database size is reduced to 150 Mbytes, which is even less I/O bound, the experiments show trends similar to those we described in this section. When $w = 10$ or larger, similar trends are observed when the system is running at the non-I/O bound status.

4.3 Sensitivity to Workload/System Parameters

The performance edge of HTSTE over Berkeley DB is most pronounced when there is a "medium" amount of physical disk I/O (i.e., when fewer than 25% transactions perform disk I/O). Figure 4 shows the performance difference between HTSTE and Berkeley DB under different cache sizes. To tune the effective database buffer cache size and thus disk I/O rate, we locked down a certain amount of main memory. Figure 5 gives the breakdown for lock waiting and average I/O time for `neword` transactions as a function of cache size. HTSTE experiences about the same amount of disk I/O as Berke-

Cache Size (Mbytes)	Neword	Payment	Slev	Ostat	Delivery
128	99.8%	71.4%	93.1%	82.3%	98.7%
256	75.5%	21.1%	87.5%	48.1%	97.3%
384	22.0%	3.7%	62.0%	18.1%	84.1%
512	13.3%	1.3%	51.0%	9.4%	71.7%

Table 4: Percentage of transactions that perform I/O during their lifetime with different cache size. The number of concurrent transactions is four. **neword**, **payment**, **slev**, **ostat**, **delivery** are five types of transactions in TPC-C. The percentages are relative to the total of each type of transaction.

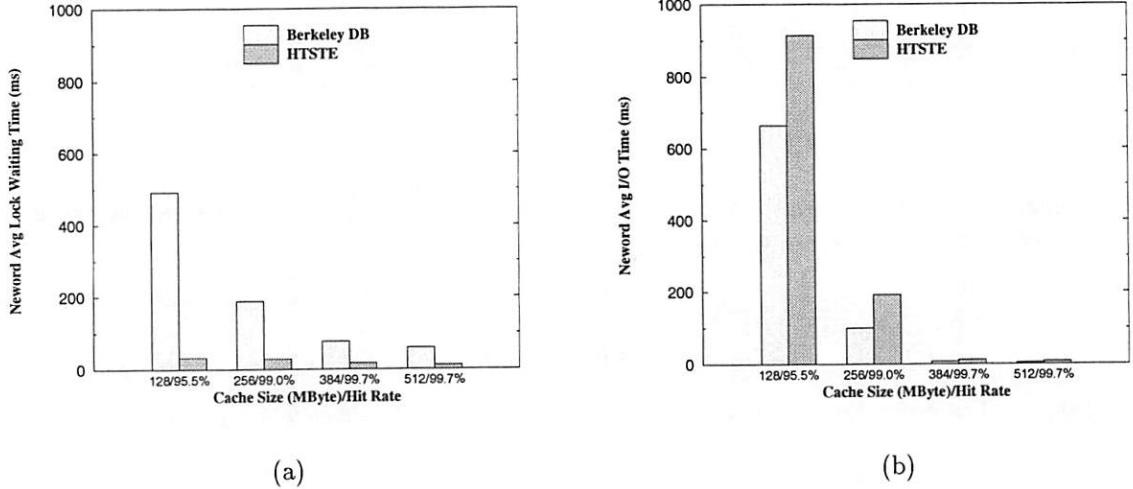


Figure 5: Lock waiting time (a) and average I/O time (b) per **neword** transaction under different cache size/hit rate.

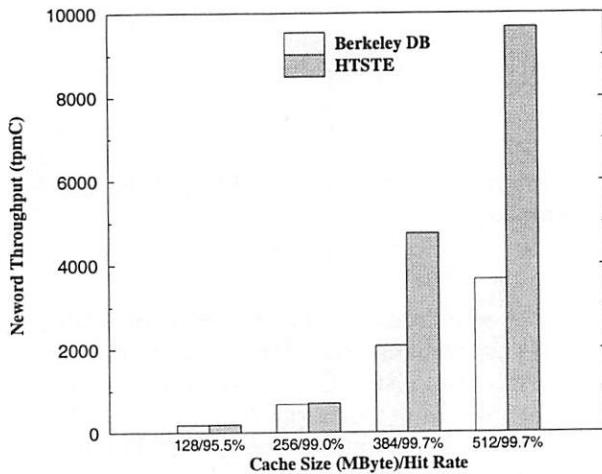


Figure 4: System throughput for **neword** transactions under different cache size/buffer hit rate. Concurrency level is 4.

ley DB. But it suffers more from disk queuing hence longer average I/O time (Figure 5(b)). When the cache hit rate is low, the storage subsystem is the performance bottleneck. The overall throughput is not improved much because the lock waiting time is now transferred to I/O queuing time. Table 5 shows that average file system read/write time for HTSTE is longer than Berkeley DB. The conditions under which TSTE algorithms excel are: some transaction that could proceed without I/O is blocked by some transaction that is holding a lock and doing I/O. If the buffer hit rate is low, the scenario becomes: most transactions need to perform I/O requests during execution, the storage system is stressed so much that I/O queuing is as bad as the cost of lock waiting. This is not the case TSTE can optimize. If the storage system can weather more burst I/O requests, then TSTE can start outperforming 2PL at a lower buffer hit rate.

Table 4 lists the percentage of transactions that need disk I/O under different cache sizes. In

Cache Size (Mbytes)	128	256	384	512
Berkeley DB	112.42	114.14	34.01	17.85
HTSTE	146.94	122.92	46.81	32.22

Table 5: Average file system read/write time comparison with different cache size. The values are in terms of msec. Concurrency level is four.

Concurrency	2	3	4	5	6
w = 1	0.9%	2.1%	2.9%	3.8%	4.4%
w = 5	1.7%	2.1%	2.4%	3.0%	3.3%

Table 6: Percentage of OTSTE transactions that need to perform physical disk I/O in the *operate* stage for varying numbers of concurrent transactions.

HTSTE, relatively less time is spent in lock waiting. As the cache size increases, the database buffer hit ratio improves, and the workload becomes less and less I/O-bound. When the workload is I/O-bound, the disk I/O time dominates the entire transaction response time, and the reduction in lock contention delay that HTSTE enables plays a less significant role in relative terms. On the other hand, when the workload is CPU-bound, there is not much physical disk I/O and HTSTE is not able to achieve a significant reduction in lock contention delays because they are relatively small to begin with. In practice, it is technically impossible for applications that require high transaction processing throughput, i.e., on the order of 1000 transactions per second, to run at an operating point that is I/O-bound. At the same time, it is economically infeasible to eliminate all disk I/Os by having enough memory to hold the entire database. Therefore, we believe that the operating point, at which high-performance transaction processing systems can achieve 1000+ transactions per second, is exactly where a TSTE-like strategy is most useful.

increases with the degree of lock contention, which in turn grows with the number of concurrent transactions. As the size of the underlying database increases (from $w = 1$ to $w = 5$), the probability of true data contention among a fixed number of concurrent transactions decreases, and so does the probability of mis-prefetching. In general, the absolute percentage of mis-prefetching transactions is quite low for the TPC-C benchmark, under 4.5% for the $w = 1$ case and under 3.4% for the $w = 5$ case. These results conclusively demonstrate that TSTE's performance cost due to mis-prefetching is insignificant for the TPC-C benchmark, and explain why TSTE out-performs Berkeley DB.

Compared to Berkeley DB, TSTE incurs additional processing overhead because it executes a transaction twice. TPC-C benchmark's five transactions all perform much more disk I/O than computation. With a benchmark requiring more computation than TPC-C, TSTE will pay more processing overhead. OTSTE corrects this problem by avoiding this additional processing overhead when all the data pages that a transaction needs are already resident in memory. To evaluate the impact of CPU time on the performance comparison among TSTE, OTSTE and Berkeley DB, we added an idle loop to the end of each TPC-C transaction, and measured the total elapsed time of completing a sequence of 10,000 transactions. By adding an idle loop into each TPC-C transaction, we simulate transactions with larger computation versus disk I/O rate. Table 7 shows the performance comparison among TSTE, HTSTE, OTSTE and Berkeley DB in terms of transaction throughput and average lock waiting time for the following two cases: a 0-msec idle loop and a 30-msec idle loop. When the transaction CPU time is small (0-msec case), TSTE's additional processing overhead does not cause serious perfor-

4.4 Detailed Analysis

One potential performance problem associated with TSTE and its variants is *mis-prefetching*, which causes transactions in the *operate* stage to perform physical disk I/Os, and thus prolong lock contention delay. Table 6 shows the average percentage of transactions that actually need to perform physical disk I/Os in their *operate* stage for both $w = 1$ and $w = 5$ cases. The probability of mis-prefetching

Idle Loop	System Throughput (tpmC)				Average Lock Waiting Time (msec)			
	TSTE	HTSTE	OTSTE	Berkeley DB	TSTE	HTSTE	OTSTE	Berkeley DB
0 msec	9662	9686	6869	3648	11.8	11.2	20.5	58.9
30 msec	7490	6507	5787	3650	14.7	16.3	26.2	56.6

Table 7: System throughput and average lock waiting time for `neword` transaction with different idle loop padding. $w = 5$. Buffer hit rate is 99.7%. Concurrency is four.

Transaction Type	Neword	Payment	Slev	Ostat	Delivery
Memory Usage ($w = 1$)	184	48	1340	64	344
Memory Usage ($w = 5$)	370	66	2305	104	534

Table 8: The average physical memory requirement for each transaction instance of the five types of transactions in the TCP-C benchmark. All numbers are in terms of Kbytes.

mance problem, and therefore TSTE always outperforms Berkeley DB. On the other hand, the optimistic optimization of OTSTE is not particularly useful when compared to TSTE or HTSTE in this case. Also, HTSTE and TSTE have similar performance. However, when the transaction CPU time is high (30-msec case), the performance difference between TSTE and Berkeley DB decreases because TSTE’s additional processing overhead erodes a significant portion of its performance gain from lock contention delay reduction. In this case, OTSTE performs worse than TSTE and HTSTE, as the cost reduction in redundant execution cost does not compensate for the additional lock contention overhead it adds. OTSTE in general requires more lock acquisitions/releases than TSTE because those lock acquisitions/releases made optimistically need to be performed twice when physical disk I/O and thus rollback occurs. HTSTE exhibits a similar problem, but to a lesser degree, and thus shows worse throughput than TSTE. A similar trend is observed in the $w = 1$ configuration.

When data is brought into main memory in the *fetch* stage TSTE pins down those buffer pages to ensure that they remain available in the *operate* stage. Table 8 shows the average physical memory usage of each transaction type in TPC-C, and shows that the maximum physical memory requirement for a TSTE-based system running 100 transactions simultaneously is about 230 Mbytes, which is relatively modest for state-of-the-art server-class machines. Therefore the additional memory pressure that TSTE introduces is less of an issue in practice.

5 Conclusion

This paper presents the design, implementation, and evaluation of a novel transaction execution engine called *Charm* that attempts to reduce the lock contention delay due to disk I/O to the minimum. The basic idea is to separate disk I/O from lock acquisition/release so that a transaction is not allowed to compete for locks unless all its required pages are guaranteed to be in memory. With this execution strategy, the lock contention delay and thus the response time of individual transactions are significantly reduced. The total elapsed time for completing a given number of transactions also improves accordingly. Specifically, this paper makes the following contributions:

- We have developed a general two-phase transaction execution (TSTE) scheme to minimize the performance impact of disk I/O on lock contention, and several of its variants to address TSTE’s redundant computation and unnecessary rollback problems.
- The TSTE prototype is the first known implementation of a two-phase transaction execution scheme that effectively decouples lock contention from physical disk I/O.
- Empirical performance measurements of the TCP-C benchmark on a working TSTE prototype demonstrate that the best variant of TSTE, HTSTE, can out-perform standard 2PL implementation by up to 164% in terms of transaction throughput, and the optimistic version of TSTE (OTSTE) actually performs worse than generic TSTE and Hybrid TSTE because extra lock contention overhead for OTSTE exceeds the saving in redundant trans-

action computation when the computation time is comparable or less than disk I/O time.

- TSTE best fits those applications with “medium” physical disk I/O. It cannot improve the performance a lot when storage system is the performance bottleneck.

One performance aspect of transaction processing systems that this research ignores is the I/O overhead associated with transaction commits. While it is a standard industry practice to reduce the commit I/O cost using group commits, the batch size chosen is typically much smaller than is used in this work. We have developed a track-based logging scheme [16] to minimize the performance impacts of the disk I/Os resulting from transaction commits, and plan to integrate TSTE with track-based logging to solve the performance problems of both read and write I/Os in high-performance transaction processing systems.

6 Acknowledgments

We would like to thank Wee Teck Ng, our shepherd Margo Seltzer, and our anonymous reviewers for their valuable feedbacks. We also appreciate the technical support team of Sleepycat Software for replying to our questions regarding the Berkeley DB package.

References

- [1] Berkeley DB package, <http://www.sleepycat.com>.
- [2] Seltzer, M.; Olson, M., “LIBTP: Portable, Modular Transactions for Unix,” *Proceedings of the Winter 1992 USENIX Conference*, p. 9-25, San Francisco, CA, USA 20-24 Jan. 1992.
- [3] Bernstein, A.J.; Wai-Hong Leung; Gerstl, D.S.; Lewis, P.M., “Design and performance of an assertional concurrency control system, *Proceedings 14th International Conference on Data Engineering*, p. 436-45, Orlando, FL, USA 23-27 Feb. 1998.
- [4] Badrinath, B.R.; Ramamirtham, K., “Synchronizing transactions on objects,” *IEEE Transactions on Computers*, vol.37, no.5, p. 541-7, May 1988.
- [5] Herlihy, M., “Apologizing versus asking permission: optimistic concurrency control for abstract data types,” *ACM Transactions on Database Systems*, vol.15, no.1, p. 96-124, March 1990.
- [6] Kung, H.T.; Robinson, J.T., “On optimistic methods for concurrency control,” *ACM Transactions on Database Systems*, vol.6, no.2, p. 213-26, June 1981.
- [7] Smith, J.E.; Weiss, S.; Pang, N.Y., “A simulation study of decoupled architecture computers,” *IEEE Transactions on Computers*, vol.C-35, no.8, p. 692-702, Aug. 1986.
- [8] Mitra, T.; Yang, C.K., “File system extensions for application-specific disk prefetching,” ECSL-TR-64, Computer Science Department, SUNY at Stony Brook, January 1999.
- [9] Jung-Ho Ahn; Hyoung-Joo Kim, “SEOF: an adaptable object prefetch policy for object-oriented database systems,” *Proceedings 13th International Conference on Data Engineering*, p. 4-13, Birmingham, UK. April 7-11, 1997.
- [10] Kimbrel, T.; Cao, P.; Felten, E.W.; Karlin, A.R.; Li, K., “Integrated parallel prefetching and caching,” *Performance Evaluation Review*, vol.24, no.1, p. 262-3, ACM May 1996.
- [11] Lehman, P.L.; Yao, S.B., “Efficient locking for concurrent operations on B-trees,” *ACM Transactions on Database Systems*, vol.6, no.4, p. 650-70, Dec. 1981.
- [12] Franaszek, P.A. et al. “Concurrency control for high contention environments,” *ACM Transactions on Database Systems*, June 1992. vol.17, no.2, p. 304-45.
- [13] Franaszek, P.A. et al. “Access invariance and its use in high contention environments,” *Proceedings of Sixth International Conference on Data Engineering*, p. 47-55, Los Angeles, CA, USA 5-9 Feb. 1990.
- [14] Reuter, A., “The transaction pipeline processor,” *Proceedings of the International Workshop on High Performance Transaction Systems*, Pacific Grove, CA, Sep. 1985.

- [15] Chang, F.; Gibson, G.A., "Automatic I/O hint generation through speculative execution," *Proceedings of Third Symposium on Operating Systems Design and Implementation*, p. 1-14, New Orleans, LA, USA 22-25 Feb. 1999.
- [16] Chiueh, T.; Huang, L.; "Trail: A Fast Synchronous Write Disk Subsystem Using Track-based Logging," ECSL-TR-68, Computer Science Department, SUNY at Stony Brook, June 1999.
- [17] Olson M.; Bostic K.; Seltzer M., "Berkeley DB," *Proceedings of the 1999 Summer Usenix Technical Conference*, Monterey, California, June 1999.

Fast Indexing: Support for Size-Changing Algorithms in Stackable File Systems

Erez Zadok
SUNY at Stony Brook
ezk@cs.sunysb.edu

Johan M. Andersen, Ion Bădulescu, and Jason Nieh
Columbia University
{johan,ion,nieh}@cs.columbia.edu

Abstract

Stackable file systems can provide extensible file system functionality with minimal performance overhead and development cost. However, previous approaches provide only limited functionality. In particular, they do not support size-changing algorithms (SCAs), which are important and useful for many applications such as compression and encryption. We propose fast indexing, a technique for efficient support of SCAs in stackable file systems. Fast indexing provides a page mapping between file system layers in a way that can be used with any SCA. We use index files to store this mapping. Index files are designed to be recoverable if lost and add less than 0.1% disk space overhead. We have implemented fast indexing using portable stackable templates, and we have used this system to build several example file systems with SCAs. We demonstrate that fast index files have low overhead for typical user workloads such as large compilations, only 2.3% over other stacked file systems and 4.7% over non-stackable file systems. Our system can deliver better performance with SCAs than user-level applications, as much as five times faster.

1 Introduction

Size-changing algorithms (SCAs) are those that take as input a stream of data bits and produce output of a different number of bits. These SCAs share one quality in common: they are generally intended to work on whole streams of input data, from the beginning to the end of the stream. Some of the applications of such algorithms fall into several possible categories:

Compression: Algorithms that reduce the overall data size to save on storage space or transmission bandwidths.

Encoding: Algorithms that encode the data such that it has a better chance of being transferred, often via email, to its intended recipients. For example, Uuencode is an algorithm that uses only the simplest printable ASCII

characters and no more than 72 characters per line. In this category we also consider transformations to support internationalization of text as well as unicoding.

Encryption: These are algorithms that transform the data so it is more difficult to decode it without an authorization—a decryption key. Encryption algorithms can work in various modes, some of which change the data size while some modes do not [23]. Typically, encryption modes that increase data size are also more secure.

There are many useful user-level tools that use SCAs, such as uuencode, compress, and pgp. These tools work on whole files and are often used manually by users. This poses additional inconvenience to users. When you encrypt or decompress a data file, even if you wish to access just a small part of that file, you still have to encode or decode all of it until you reach the portion of interest—an action that consumes many resources. SCAs do not provide information that can help to decode or encode only the portion of interest. Furthermore, running user-level SCA tools repeatedly costs in additional overhead as data must be copied between the user process and the kernel several times. User-level SCA tools are therefore neither transparent to users nor do they perform well.

Instead, it would be useful for a file system to support SCAs. File systems are (1) transparent to users since they do not have to run special tools to use files, and (2) perform well since they often run in the kernel. File systems have proven to be a useful abstraction for extending system functionality. Several SCAs (often compression) have been implemented as extensions to existing disk-based file systems [2, 3, 18]. Their disadvantages are that they only work with specific operating systems and file systems, and they only support those specific SCAs. Supporting general-purpose SCAs on a wide range of platforms was not possible.

Stackable file systems are an effective infrastructure for creating new file system functionality with minimal performance overhead and development cost [10, 12, 22, 24, 28,

29, 25]. Stackable file systems can be developed independently and then stacked on top of each other to provide new functionality. Also, they are more portable and are easier to develop [29]. For example, an encryption file system can be mounted on top of a native file system to provide secure and transparent data storage [27]. Unfortunately, general-purpose SCAs have never been implemented in stackable file systems. The problem we set out to solve was how to support general-purpose SCAs in a way that is easy to use, performs well, and is available for many file systems.

We propose *fast indexing* as a solution for supporting SCAs in stackable file systems. Fast indexing provide a way to map file offsets between upper and lower layers in stackable file systems. Since the fast indexing is just a mapping, a lower-layer file system does not have to know anything about the details of the SCA used by an upper-level file system. We store this fast indexing information in *index files*. Each encoded file has a corresponding index file which is simply stored in a separate file in the lower-layer file system. The index file is much smaller than the original data file, resulting in negligible storage requirements. The index file is designed to be recoverable if it is somehow lost so that it does not compromise the reliability of the file system. Finally, fast indexing is designed to deliver good file system performance with low stacking overhead, especially for common file operations.

We have implemented fast indexing using stackable templates [28, 29, 25]. This allows us to provide transparent support for SCAs in a portable way. To demonstrate the effectiveness of our approach, we built and tested several size-changing file systems, including a compression file system. Our performance results show (1) that fast index files have low overhead for typical file system workloads, only 2.3% over other null-layer stackable file systems, and (2) that such file systems can deliver as much as five times better performance than user-level SCA applications.

This paper describes fast index files and is organized as follows. Section 2 reviews the stacking file-system infrastructure used for this work and discusses related work in SCA support in file systems. Section 3 details the design of the index file. Section 4 describes the usage of the index file in relation to common file operations and discusses several optimizations. Section 5 details our design for a consistent and recoverable index file. Section 6 summarizes important implementation issues. Section 7 describes the file systems we built using this work and evaluates our system. Finally, we present conclusions and discuss directions for future work.

2 Background

In this section we discuss extensibility mechanisms for file systems, what would be required for such file systems to support SCAs, and other systems that provide some support

for compression SCAs.

2.1 Stacking Support

Stackable file systems allow for modular, incremental development of file systems by layering additional functionality on another file system [13, 15, 21, 24]. Stacking provides an infrastructure for the composition of multiple file systems into one.

Kernel User

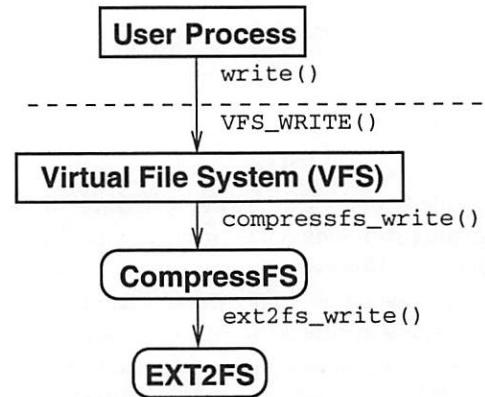


Figure 1: An example stackable compression file system. A system call is translated into a generic VFS function, which is translated into a file-system specific function in our stackable compression file system. CompressFS then modifies (compresses) the data passed to it and calls the file system stacked below it with the modified data.

Figure 1 shows the structure for a simple single-level stackable compression file system called CompressFS. System calls are translated into VFS calls, which in turn invoke their CompressFS equivalents. CompressFS receives user data to be written. It compresses the data and passes it to the next lower layer, without any regard to what type of file system implements that layer.

Stackable file systems were designed to be modular and transparent: each layer is independent from the layers above and below it. In that way, stackable file system modules could be composed together in different configurations to provide new functionality. Unfortunately, this poses problems for SCAs because the decoded data at the upper layer has different file offsets from the encoded data at the lower layer. CompressFS, for example, must know how much compressed data it wrote, where it wrote it, and what original offsets in the decoded file did that data represent. Those pieces of information are necessary so that subsequent reading operations can locate the data quickly. If CompressFS cannot find the data quickly, it may have to resort to decompression of the complete file before it can locate the data to read.

Therefore, to support SCAs in stackable file systems, a stackable layer must have some information about the en-

coded data—offset information. But a stackable file system that gets that information about other layers violates its transparency and independence. This is the main reason why past stacking works do not support SCAs. The challenge we faced was to add general-purpose SCA support to a stacking infrastructure without losing the benefits of stacking: a stackable file system with SCA support should not have to know anything about the file system it stacks on. That way it can add SCA functionality automatically to any other file system.

2.2 Compression Support

Compression file systems are not a new idea. Windows NT supports compression in NTFS [18]. E2compr is a set of patches to Linux’s Ext2 file system that add block-level compression [2]. Compression extensions to log-structured file systems resulted in halving of the storage needed while degrading performance by no more than 60% [3]. The benefit of block-level compression file systems is primarily speed. Their main disadvantage is that they are specific to one operating system and one file system, making them difficult to port to other systems and resulting in code that is hard to maintain.

The ATTIC system demonstrated the usefulness of automatic compression of least-recently-used files [5]. It was implemented as a modified user-level NFS server. Whereas it provided portable code, in-kernel file systems typically perform better. In addition, the ATTIC system decompresses whole files which slows performance.

HURD [4] and Plan 9 [19] have an extensible file system interface and have suggested the idea of stackable compression file systems. Their primary focus was on the basic minimal extensibility infrastructure; they did not produce any working examples of size-changing file systems.

Spring [14, 16] and Ficus [11] discussed a similar idea for implementing a stackable compression file system. Both suggested a unified cache manager that can automatically map compressed and uncompressed pages to each other. Heidemann’s Ficus work provided additional details on mapping cached pages of different sizes.¹ Unfortunately, no demonstration of these ideas for compression file systems was available from either of these works. In addition, no consideration was given to arbitrary SCAs and how to efficiently handle common file operations such as appends, looking up file attributes, etc.

¹Heidemann’s earlier work [13] mentioned a “prototype compression layer” built during a class project. In personal communications with the author, we were told that this prototype was implemented as a block-level compression file system, not a stackable one.

3 The Index File

In a stacking environment that supports SCAs, data offsets may change arbitrarily. An efficient mapping is needed that can tell where the starting offset of the encoded data is for a given offset in the original file. We call this mapping the *index table*.

The index table is stored in a separate file called the *index file*, as shown in Figure 2. This file serves as the fast metadata index into an encoded file. For a given data file F , we create an index file called $F.idx$. Many file systems separate data and meta data; this is done for efficiency and reliability. Meta-data is considered more important and so it gets cached, stored, and updated differently than regular data. The index file is separate from the encoded file data for the same reasons and to allow us to manage each part separately and simply.

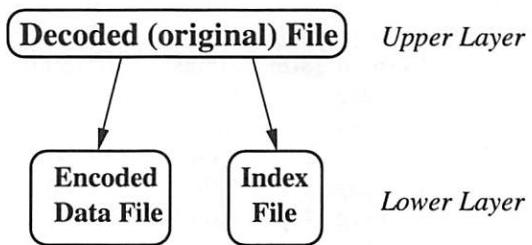


Figure 2: Overall structure of size-changing stackable file systems. Each original data file is encoded into a lower data file. Additional meta-data index information is stored in an index file. Both the index file and the encoded data files reside in the lower level file system.

Our system encodes and decodes whole pages or their multiples—which maps well to file system operations. The index table assumes page-based operations and stores offsets of encoded pages as they appear in the encoded file.

Consider an example of a file in a compression file system. Figure 3 shows the mapping of offsets between the upper (original) file and the lower (encoded) data file. To find out the bytes in page 2 of the original file, we read the data bytes 3000–7200 in the encoded data file, decode them, and return to the VFS that data in page 2.

To find out which encoded bytes we need to read from the lower file, we consult the index file, shown in Table 1. The index file tells us that the original file has 6 pages, that its original size is 21500 bytes, and then it lists the ending offsets of the encoded data for an upper page. Finding the lower offsets for the upper page 2 is a simple linear dereferencing of the data in the index file; we do not have to search the index file linearly. Note that our design of the index file supports both 32-bit and 64-bit file systems, but the examples we provide here are for 32-bit file systems.

The index file starts with a word that encodes flags and the number of pages in the corresponding original data

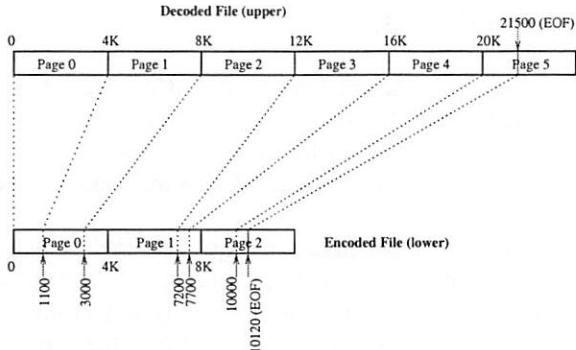


Figure 3: An example of a 32-bit file system that shrinks data size (compression). Each upper page is represented by an encoded lower “chunk.” The mapping of offsets is shown in Table 1.

Word (32/64 bits)	Representing	Regular IDX File	With Fast Tail (ft)
1 (12 bits)	flags	ls=0,ft=0,...	ls=0,ft=1,...
1 (20/52 bits)	# pages	6	5
2	orig. file size	21500	21500
3	page 0	1100	1100
4	page 1	3000	3000
5	page 2	7200	7200
6	page 3	7700	7700
7	page 4	10000	10000
8	page 5	10120	

Table 1: Format of the index file for Figures 3 and 4. Fast Tails are described in Section 4.2. The first word encodes both flags and the number of pages in the index file. The “ls” (large size) flag is the first bit in the index file and indicates if the index file encodes a 32-bit (0) or 64-bit (1) file system.

file. We reserve the lower 12 bits for special flags such as whether the index file encodes a file in a 32-bit or a 64-bit file system, whether fast tails were encoded in this file (see Section 4.2), etc. The very first bit of these flags, and therefore the first bit in the index file, determines if the file encoded is part of a 32-bit or a 64-bit file system. This way, just by reading the first bit we can determine how to interpret the rest of the index file: 4 bytes to encode page offsets on 32-bit file systems or 8 bytes to encode page offsets on 64-bit file systems.

We use the remaining 20 bits (on a 32-bit file system) for the number of pages because 2^{20} 4KB pages (the typical page size on i386 and SPARCv8 systems) would give us the exact maximum file size we can encode in 4 bytes on a 32-bit file system, as explained next; similarly 2^{52} 4KB pages is the exact maximum file size on a 64-bit file system.

The index file also contains the original file’s size in the second word. We store this information in the index file so that commands like `ls -l` and others using `stat(2)` would work correctly; a process looking at the size of the file through the upper level file system would get the origi-

nal number of bytes and blocks. The original file’s size can be computed from the starting offset of the last data chunk in the encoded file, but it would require decoding the last (possibly incomplete) chunk (bytes 10000–10120 in the encoded file in Figure 3) which can be an expensive operation depending on the SCA. Storing the original file size in the index file is a speed optimization that only consumes one more word—in a physical data block that most likely was already allocated.

The index file is small. We store one word (4 bytes on a 32-bit file system) for each data page (usually 4096 bytes). On average, the index table size is 1024 times smaller than the original data file. For example, an index file that is exactly 4096 bytes long (one disk block on an Ext2 file system formatted with 4KB blocks) can describe an original file size of 1022 pages, or 4,186,112 bytes (almost 4MB).

Since the index file is relatively small, we read it into kernel memory as soon as the main file is open and manipulate it there. That way we have fast access to the index data in memory. The index information for each page is stored linearly and each index entry typically takes 4 bytes. That lets us compute the needed index information simply and find it from the index table using a single dereference into an array of 4-byte words (integers). To improve performance further, we write the final modified index table only after the original file is closed and all of its data flushed to stable media.

The size of the index file is less important for SCAs which increase the data size, such as unicoding, uuencoding, and some forms of encryption. The more the SCA increases the data size, the less significant the size of the index file becomes. Even in the case of SCAs that decrease data size (e.g., compression) the size of the index file may not be as important given the savings already gained from compression.

Since the index information is stored in a separate file, it uses up one more inode. We measured the effect that the consumption of an additional inode would have on typical file systems in our environment. We found that disk data block usage is often 6–8 times greater than inode utilization on disk-based file systems, leaving plenty of free inodes to use. To save resources even further, we efficiently support zero-length files: a zero-length original data file is represented by a zero-length index file.

For reliability reasons, we designed the index file so it could be recovered from the data file in case the index file is lost or damaged (Section 5.) This offers certain improvements over typical Unix file systems: if their meta-data (inodes, inode and indirect blocks, directory data blocks, etc.) is lost, it rarely can be recovered. Note that the index file is not needed for our system to function: it represents a performance enhancing tool. Without the index file, size-changing file systems would perform poorly. Therefore, if it does not exist (or is lost), our system automatically re-

generates the index file.

4 File Operations

We now discuss the handling of file system operations in fast indexing as well as specific optimizations for common operations. Note that most of this design relates to performance optimizations while a small part (Section 4.4) addresses correctness.

Because the cost of SCAs can be high, it is important to ensure that we minimize the number of times we invoke these algorithms and the number of bytes they have to process each time. The way we store and access encoded data chunks can affect this performance as well as the types and frequencies of file operations. As a result, fast indexing takes into account the fact that file accesses follow several patterns:

- The most popular file system operation is `stat(2)`, which results in a file lookup. Lookups account for 40–50% of all file system operations [17, 20].
- Most files are read, not written. The ratio of reads to writes is often 4–6 [17, 20]. For example, compilers and editors read in many header and configuration files, but only write out a handful of files.
- Files that are written are often written from beginning to end. Compilers, user tools such as `cp`, and editors such as `emacs` write whole files in this way. Furthermore, the unit of writing is usually set to match the system page size. We have verified this by running a set of common edit and build tools on Linux and recorded the write start offsets, size of write buffers, and the current size of the file.
- Files that are not written from beginning to end are often appended to. The number of appended bytes is usually small. This is true for various log files that reside in `/var/log` as well as Web server logs.
- Very few files are written in the middle. This happens in two cases. First, when the GNU `ld` creates large binaries, it writes a sparse file of the target size and then seeks and writes the rest of the file in a non-sequential manner. To estimate the frequency of writes in the middle, we instrumented a null-layer file system with a few counters. We then measured the number and type of writes for our large compile benchmark (Section 7.3.1). We counted 9193 writes, of which 58 (0.6%) were writes before the end of a file.

Second, data-base files are also written in the middle. We surveyed our own site’s file servers and workstations (several hundred hosts totaling over 1TB of storage) and found that these files represented less than

0.015% of all storage. Of those, only 2.4% were modified in the past 30 days, and only 3% were larger than 100MB.

- All other operations (together) account for a small fraction of file operations [17, 20].

We designed our system to optimize performance for the more common and important cases while not harming performance unduly when the seldom-executed cases occur. We first describe how the index file is designed to support fast lookups, file reads, and whole file writes, which together are the most common basic file operations. We then discuss support for appending to files efficiently, handling the less common operation of writes in the middle of files, and ensuring correctness for the infrequent use of truncate.

4.1 Basic Operations

To handle file lookups fast, we store the original file’s size in the index table. Due to locality in the creation of the index file, we assume that its name will be found in the same directory block as the original file name, and that the inode for the index file will be found in the same inode block as the encoded data file. Therefore reading the index file requires reading one additional inode and often only one data block. After the index file is read into memory, returning the file size is done by copying the information from the index table into the “size” field in the current inode structure. Remaining attributes of the original file come from the inode of the actual encoded file. Once we read the index table into memory, we allow the system to cache its data for as long as possible. That way, subsequent lookups will find files’ attributes in the attribute cache.

Since most file systems are structured and implemented internally for access and caching of whole pages, we also encode the original data file in whole pages. This improved our performance and helped simplify our code because interfacing with the VFS and the page cache was more natural. For file reads, the cost of reading in a data page is fixed: a fixed offset lookup into the index table gives us the offsets of encoded data on the lower level data file; we read this encoded sequence of bytes, decode it into exactly one page, and return that decoded page to the user.

Using entire pages made it easier for us to write whole files, especially if the write unit was one page size. In the case of whole file writes, we simply encode each page size unit, add it to the lower level encoded file, and add one more entry to the index table. We discuss the cases of file appends and writes in the middle in Sections 4.2 and 4.3, respectively.

We did not have to design anything special for handling all other file operations. We simply treat the index file at the same time we manipulate the corresponding encoded data file. An index file is created only for regular files; we

do not have to worry about symbolic links because the VFS will only call our file system to open a regular file. When a file is hard-linked, we also hard-link the index file using the name of the new link with a the .idx extension added. When a file is removed from a directory or renamed, we apply the same operation to the corresponding index file.

4.2 Fast Tails

One common usage pattern of files is to append to them. Often, a small number of bytes is appended to an existing file. Encoding algorithms such as compression and encryption are more efficient when they encode larger chunks of data. Therefore it is better to encode a larger number of bytes together. Our design calls for encoding whole pages whenever possible. Table 1 and Figure 3 show that only the last page in the original file may be incomplete and that incomplete page gets encoded too. If we append, say, 10 more bytes to the original (upper) file of Figure 3, we have to keep it and the index file consistent: we must read the 1020 bytes from 20480 until 21500, decode them, add the 10 new bytes, encode the new 1030 sequence of bytes, and write it out in place of the older 1020 bytes in the lower file. We also have to update the index table in two places: the total size of the original file is now 21510, and word number 8 in the index file may be in a different location than 10120 (depending on the encoding algorithm, it may be greater, smaller, or even the same).

The need to read, decode, append, and re-encode a chunk of bytes for each append grows worse as the number of bytes to append is small while the number of encoded bytes is closer to one full page. In the worst case, this method yields a complexity of $O(n^2)$ in the number of bytes that have to be decoded and encoded, multiplied by the cost of the encoding and decoding of the SCA. To solve this problem, we added a *fast tails* runtime mount option that allows for up to a page size worth of unencoded data to be added to an otherwise encoded data file. This is shown in the example in Figure 4.

In this example, the last full page that was encoded is page 4. Its data bytes end on the encoded data file at offset 10000 (page 2). The last page of the original upper file contains 1020 bytes (21500 less 20K). So we store these 1020 bytes directly at the end of the encoded file, after offset 10000. To aid in computing the size of the fast tail, we add two more bytes to the end of the file past the fast tail itself, listing the length of the fast tail. (Two bytes is enough to list this length since typical page sizes are less than 2^{16} bytes long.) The final size of the encoded file is now 11022 bytes long.

With fast tails, the index file does not record the offset of the last tail as can be seen from the right-most column of Table 1. The index file, however, does record in its flags field (first 12 bits of the first word) that a fast tail is in use.

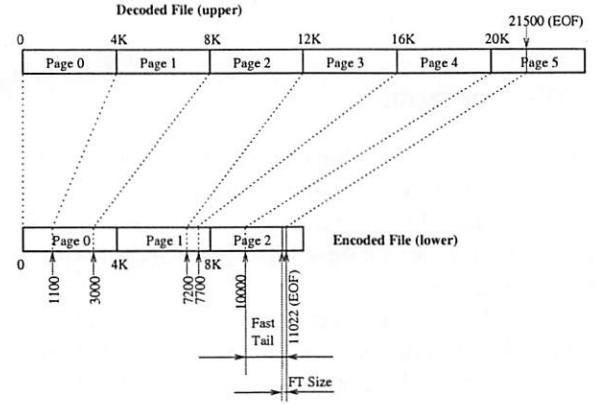


Figure 4: Size-changed file structure with fast-tail optimization. A file system similar to Figure 3, only here we store up to one page full of unencoded raw data. When enough raw data is collected to fill a whole fast-tail page, that page is encoded.

We put that flag in the index table to speed up the computations that depend on the presence of fast tails. We append the length of the fast tail to the encoded data file to aid in reconstruction of a potentially lost index file, as described in Section 5.

When fast tails are in use, appending a small number of bytes to an existing file does not require data encoding or decoding, which can speed up the append operation considerably. When the size of the fast tail exceeds one page, we encode the first page worth of bytes, and start a new fast tail.

Fast tails, however, may not be desirable all the time exactly because they store unencoded bytes in the encoded file. If the SCA used is an encryption one, it is insecure to expose plaintext bytes at the end of the ciphertext file. For this reason, fast tails is a runtime global mount option that affects the whole file system mounted with it. The option is global because typically users wish to change the overall behavior of the file system with respect to this feature, not on a per-file basis.

4.3 Write in the Middle

User processes can write any number of bytes in the middle of an existing file. With our system, whole pages are encoded and stored in a lower level file as individual encoded chunks. A new set of bytes written in the middle of the file may encode to a different number of bytes in the lower level file. If the number of new encoded bytes is greater than the old number, we shift the remaining encoded file outward to make room for the new bytes. If the number of bytes is smaller, we shift the remaining encoded file inward to cover unused space. In addition, we adjust the index table for each encoded data chunk which was shifted. We perform shift operations as soon as our file system's write

operation is invoked, to ensure sequential data consistency of the file.

To improve performance, we shift data pages in memory and keep them in the cache as long as possible: we avoid flushing those data pages to disk and let the system decide when it wants to do so. That way, subsequent write-in-the-middle operations that may result in additional inward or outward shifts will only have to manipulate data pages already cached and in memory. Any data page shifted is marked as dirty, and we let the paging system flush it to disk when it sees fit.

Note that data that is shifted in the lower level file does not have to be re-encoded. This is because that data still represents the actual encoded chunks that decode into their respective pages in the upper file. The only thing remaining is to change the end offsets for each shifted encoded chunk in the index file.

We examined several performance optimization alternatives that would have encoded the information about inward or outward shifts in the index table or possibly in some of the shifted data. We rejected them for several reasons: (1) it would have complicated the code considerably, (2) it would have made recovery of an index file difficult, and (3) it would have resulted in fragmented data files that would have required a defragmentation procedure. Since the number of writes in the middle we measured was so small (0.6% of all writes), we do consider our simplified design as a good cost vs. performance balance. Even with our simplified solution, our file systems work perfectly correctly. Section 7.3.2 shows the benchmarks we ran to test writes in the middle and demonstrates that our solution produces good overall performance.

4.4 Truncate

One design issue we faced was with the `truncate(2)` system call. Although this call occurs less than 0.02% of the time [17, 20], we still had to ensure that it behaved correctly. Truncate can be used to shrink a file as well as enlarge it, potentially making it sparse with new “holes.” We dealt with four cases:

1. Truncating on a page boundary. In this case, we truncate the encoded file exactly after the end of the chunk that now represents the last page of the upper file. We update the index table accordingly: it has fewer pages in it.
2. Truncating in the middle of an existing page. This case results in a partial page: we read and decode the whole page and re-encode the bytes within the page representing the part before the truncation point. We update the index table accordingly: it now has fewer pages in it.

3. Truncating in the middle of a fast tail. In that case we just truncate the lower file where the fast tail is actually located. We then update the size of the fast tail at its end and update the index file to indicate the (now) smaller size of the original file.
4. Truncating past the end of the file is akin to extending the size of the file and possibly creating zero-filled holes. We read and re-encode any partially filled page or fast tail that used to be at the end of the file before the truncation; we have to do that because that page now contains a mix of non-zero data and zeroed data. We encode all subsequent zero-filled pages. This is important for some applications such as encryption, where every bit of data—zeros or otherwise—should be encrypted.

5 Index File Consistency

With the introduction of a separate index file to store the index table, we now have to maintain two files consistently.

Normally, when a file is created, the directory of that file is locked. We keep both the directory and the encoded data file locked when we update the index file. This way both the encoded data file and the index file are guaranteed to be written correctly.

We assume that encoded data files and index files will not become corrupt internally due to media failures. This situation is no worse than normal file systems where a random data corruption may not be possible to fix. However, we do concern ourselves with three potential problems with the index file: partially written file, a lost file, and trivial corruptions.

An index file could be partially written if the file system is full or the user ran out of quota. In the case where we were unable to write the complete index file, we simply remove it and log a warning message through `syslog(3)`—where the message could be passed on to a centralized logging facility that monitors and generates appropriate alerts. The absence of the index file on subsequent file accesses will trigger an in-kernel mechanism to recover the index file. That way the index file is not necessary for our system to function; it only aids in improving performance.

An index file could be lost if it was removed intentionally (say after a partial write) or unintentionally by a user directly from the lower file system. If the index file is lost or does not exist, we can no longer easily tell where encoded bytes were stored. In the worst case, without an index file, we have to decode the complete file to locate any arbitrary byte within. However, since the cost of decoding a complete file and regenerating an index table are nearly identical (see Section 7.6), we chose to regenerate the index table immediately if it does not exist, and then proceed

as usual as the index file now exists.

We verify the validity of the index file when we use the index table. We check that all index entries are monotonically increasing, that it has the correct number of entries, file size matches the last entry, flags used are known, etc. The index file is regenerated if an inconsistency is detected. This helps our system to survive certain meta-data corruptions that could occur as a result of software bugs or direct editing of the index file.

We designed our system so that the index file can be recovered reliably in all cases. Four important pieces of information are needed to recover an index file given an encoded data file. These four are available in the kernel to the running file system:

1. The SCA used.
2. The page size of the system on which the encoded data file was created.
3. Whether the file system used is 32-bit or 64-bit.
4. Whether fast tails were used.

To recover an index file we read an input encoded data file and decode the bytes until we fill out one whole page of output data. We rely on the fact that the original data file was encoded in units of page size. The offset of the input data where we finished decoding onto one full page becomes the first entry in the index table. We continue reading input bytes and produce more full pages and more index table entries. If fast tails were used, then we read the size of the fast tail from the last two bytes of the encoded file, and we do not try to decode it since it was written unencoded.

If fast tails were not used and we reached the end of the input file, that last chunk of bytes may not decode to a whole output page. In that case, we know that was the end of the original file, and we mark the last page in the index table as a partial page. While we are decoding pages, we sum up the number of decoded bytes and fast tails, if any. The total is the original size of the data file, which we record in the index table. We now have all the information necessary to write the correct index file and we do so.

6 SCA Implementation

Our SCA support was integrated into FiST [29, 25]. The FiST system includes portable stackable file system templates for several operating systems as well as a high-level language that can describe new stackable file systems [26, 28]. Most of the work was put into the stackable templates, where we added substantially more code to support SCAs: 2119 non-comment lines of C code, representing a 60% increase in the size of the templates. Because this additional code is substantial and carries an overhead with it

that is not needed for non-size-changing file systems (Section 7), we made it optional. To support that, we added one additional declaration to the FiST language, to allow developers to decide whether or not to include this additional support.

To use FiST to produce a size-changing file system, developers need to include a single FiST declaration in their input file and then write only two routines: `encode_data` and `decode_data`. The main advantage of using FiST for this work has been the ease of use for developers that want to write size-changing file systems. All the complexity is placed in the templates and is mostly hidden from developers' view. Developers need only concentrate on the core implementation issues of the particular algorithm they wish to use in their new file system.

The FiST system has been ported to Linux, Solaris, and FreeBSD. Current SCA support is available for Linux 2.3 only. Our primary goal in this work was to prove that size-changing stackable file systems can be designed to perform well. When we feel that the design is stable and addresses all of the algorithmic issues related to the index file, we will port it to the other templates. We would then be able to describe an SCA file system once in the FiST language; from this single portable description, we could then produce a number of working file systems.

There are two implementation-specific issues of interest: one concerning Linux and the other regarding writes in the middle of files. As mentioned in Section 3, we write any modified index information out when the main file is closed and its data flushed to stable media. In Linux, neither data nor meta-data are automatically flushed to disk. Instead, a kernel thread (`kflushd`) runs every 5 seconds and asks the page cache to flush any file system data that has not been used recently, but only if the system needs more memory. In addition, file data is forced to disk when either the file system is unmounted or the process called an explicit `fflush(3)` or `fsync(2)`. We take advantage of this delayed write to improve performance, since we write the index table when the rest of the file's data is written.

To support writes in the middle correctly, we have to make an extra copy of data pages into a temporary location. The problem is that when we write a data page given to us by the VFS, we do not know what this data page will encode into, and how much space that new encoding would require. If it requires more space, then we have to shift data outward in the encoded data file before writing the new data. For this first implementation, we chose the simplified approach of always making the temporary copy, which affects performance as seen in Section 7. While our code shows good performance, it has not been optimized much yet; we discuss avenues of future work in Section 9.

7 Evaluation

To evaluate fast indexing in a real world operating system environment, we built several SCA stackable file systems based on fast indexing. We then conducted extensive measurements in Linux comparing them against non-SCA file systems on a variety of file system workloads. In this section we discuss the experiments we performed on these systems to (1) show overall performance on general-purpose file system workloads, (2) determine the performance of individual common file operations and related optimizations, and (3) compare the efficiency of SCAs in stackable file systems to equivalent user-level tools. Section 7.1 describes the SCA file systems we built and our experimental design. Section 7.2 describes the file system workloads we used for our measurements. Sections 7.3 to 7.6 present our experimental results.

7.1 Experimental Design

We ran our experiments on five file systems. We built three SCA file systems and compared their performance to two non-SCA file systems. The three SCA file systems we built were:

1. **Copyfs**: this file system simply copies its input bytes to its output without changing data sizes. Copyfs exercises all of the index-management algorithms and other SCA support without the cost of encoding or decoding pages.
2. **Uuencodefs**: this is a file system that stores files in uuencoded format and uudecodes files when they are read. It is intended to illustrate an algorithm that increases the data size. This simple algorithm converts every 3-byte sequence into a 4-byte sequence. Uuencode produces 4 bytes that can have at most 64 values each, starting at the ASCII character for space (20_h). We chose this algorithm because it is simple and yet increases data size significantly (by one third).
3. **Gzipfs**: this is a compression file system using the Deflate algorithm [7] from the zlib-1.1.3 package [9]. This algorithm is used by GNU zip (gzip) [8]. This file system is intended to demonstrate an algorithm that (usually) reduces data size.

The two non-SCA file systems we used were Ext2fs, the native disk-based file system most commonly used in Linux, and Wrapfs, a stackable null-layer file system we trivially generated using FiST [25, 29]. Ext2fs provides a measure of base file system performance without any stacking or SCA overhead. Wrapfs simply copies the data of files between layers but does not include SCA support. By comparing Wrapfs to Ext2fs, we can measure the overhead of stacking and copying data without fast indexing

and without changing its content or size. Copyfs copies data like Wrapfs but uses all of the SCA support. By comparing Copyfs to Wrapfs, we can measure the overhead of basic SCA support. By comparing Uuencodefs to Copyfs, we can measure the overhead of an SCA algorithm incorporated into the file system that increases data size. Similarly, by comparing Gzipfs to Copyfs, we can measure the overhead of a compression file system that reduces data size.

One of the primary optimizations in this work is fast tails as described in Section 4.2. For all of the SCA file systems, we ran all of our tests first without fail-tails support enabled and then with it. We reported results for both whenever fast tails made a difference.

All experiments were conducted on four equivalent 433Mhz Intel Celeron machines with 128MB of RAM and a Quantum Fireball lct10 9.8GB IDE disk drive. We installed a Linux 2.3.99-pre3 kernel on each machine. Each of the four stackable file systems we tested was mounted on top of an Ext2 file system. For each benchmark, we only read, wrote, or compiled the test files in the file system being tested. All other user utilities, compilers, headers, and libraries resided outside the tested file system.

Unless otherwise noted, all tests were run with a cold cache. To ensure that we used a cold cache for each test, we unmounted all file systems which participated in the given test after the test completed and mounted the file systems again before running the next iteration of the test. We verified that unmounting a file system indeed flushes and discards all possible cached information about that file system. In one benchmark we report the warm cache performance, to show the effectiveness of our code's interaction with the page and attribute caches.

We ran all of our experiments 10 times on an otherwise quiet system. We measured the standard deviations in our experiments and found them to be small, less than 1% for most micro-benchmarks described in Section 7.2. We report deviations which exceeded 1% with their relevant benchmarks.

7.2 File System Benchmarks

We measured the performance of the five file systems on a variety of file system workloads. For our workloads, we used five file system benchmarks: two general-purpose benchmarks for measuring overall file system performance, and three micro-benchmarks for measuring the performance of common file operations that may be impacted by fast indexing. We also used the micro-benchmarks to compare the efficiency of SCAs in stackable file systems to equivalent user-level tools.

7.2.1 General-Purpose Benchmarks

Am-utils: The first benchmark we used to measure overall file system performance was am-utils (The Berkeley Automounter) [1]. This benchmark configures and compiles the large am-utils software package inside a given file system. We used am-utils-6.0.4: it contains over 50,000 lines of C code in 960 files. The build process begins by running several hundred small configuration tests intended to detect system features. It then builds a shared library, about ten binaries, four scripts, and documentation: a total of 265 additional files. Overall this benchmark contains a large number of reads, writes, and file lookups, as well as a fair mix of most other file system operations such as unlink, mkdir, and symlink. During the linking phase, several large binaries are linked by GNU ld.

The am-utils benchmark is the only test that we also ran with a warm cache. Our stackable file systems cache decoded and encoded pages whenever possible, to improve performance. While normal file system benchmarks are done using a cold cache, we also felt that there is value in showing what effect our caching has on performance. This is because user level SCA tools rarely benefit from page caching, while file systems are designed to perform better with warm caches; this is what users will experience in practice.

Bonnie: The second benchmark we used to measure overall file system performance was Bonnie [6], a file system test that intensely exercises file data reading and writing, both sequential and random. Bonnie is a less general benchmark than am-utils. Bonnie has three phases. First, it creates a file of a given size by writing it one character at a time, then one block at a time, and then it rewrites the same file 1024 bytes at a time. Second, Bonnie writes the file one character at a time, then a block at a time; this can be used to exercise the file system cache, since cached pages have to be invalidated as they get overwritten. Third, Bonnie forks 3 processes that each perform 4000 random 1 seeks in the file, and read one block; in 10% of those seeks, Bonnie also writes the block with random data. This last phase exercises the file system quite intensively, and especially the code that performs writes in the middle of files.

For our experiments, we ran Bonnie using files of increasing sizes, from 1MB and doubling in size up to 128MB. The last size is important because it matched the available memory on our systems. Running Bonnie on a file that large is important, especially in a stackable setting where pages are cached in both layers, because the page cache should not be able to hold the complete file in memory.

7.2.2 Micro-Benchmarks

File-copy: The first micro-benchmark we used was designed to measure file system performance on typical bulk

file writes. This benchmark copies files of different sizes into the file system being tested. Each file is copied just once. Because file system performance can be affected by the size of the file, we exponentially varied the sizes of the files we ran these tests on—from 0 bytes all the way to 32MB files.

File-append: The second micro-benchmark we used was designed to measure file system performance on file appends. It was useful for evaluating the effectiveness of our fast tails code. This benchmark read in large files of different types and used their bytes to append to a newly created file. New files are created by appending to them a fixed but growing number of bytes. The benchmark appended bytes in three different sizes: 10 bytes representing a relatively small append; 100 bytes representing a typical size for a log entry on a Web server or syslog daemon; and 1000 bytes, representing a relatively large append unit. We did not try to append more than 4KB because that is the boundary where fast appended bytes get encoded. Because file system performance can be affected by the size of the file, we exponentially varied the sizes of the files we ran these tests on—from 0 bytes all the way to 32MB files.

Compression algorithms such as used in Gzipfs behave differently based on the input they are given. To account for this in evaluating the append performance of Gzipfs, we ran the file-append benchmark on four types of data files, ranging from easy to compress to difficult to compress:

1. A file containing the character “a” repeatedly should compress really well.
2. A file containing English text, actually written by users, collected from our Usenet News server. We expected this file to compress well.
3. A file containing a concatenation of many different binaries we located on the same host system, such as those found in /usr/bin and /usr/X11R6/bin. This file should be more difficult to compress because it contains fewer patterns useful for compression algorithms.
4. A file containing previously compressed data. We took this data from Microsoft NT’s Service Pack 6 (sp6i386.exe) which is a self-unarchiving large compressed executable. We expect this file to be difficult to compress.

File-attributes: The third micro-benchmark we used was designed to measure file system performance in getting file attributes. This benchmark performs a recursive listing (ls -lRF) on a freshly unpacked and built am-utils benchmark file set, consisting of 1225 files. With our SCA support, the size of the original file is now stored in the index file, not in the inode of the encoded data file. Finding this size requires reading an additional inode of the index

file and then reading its data. This micro-benchmark measures the additional overhead that results from also having to read the index file.

7.2.3 File System vs. User-Level Tool Benchmarks

To compare the SCAs in our stackable file systems versus user-level tools, we used the file-copy micro-benchmark to compare the performance of the two stackable file systems with real SCAs, Gzipfs and Uuencodefs, against their equivalent user-level tools, *gzip* [8] and *uuencode*, respectively. In particular, the same Deflate algorithm and compression level (9) was used for both Gzipfs and *gzip*. In comparing Gzipfs and *gzip*, we measured both the compression time and the resulting space savings. Because the performance of compression algorithms depends on the type of input, we compared Gzipfs to *gzip* using the file-copy micro-benchmark on all four of the different file types discussed in Section 7.2.2.

7.3 General-Purpose Benchmark Results

7.3.1 Am-Utils

Figure 5 summarizes the results of the am-utils benchmark. We report both system and elapsed times. The top part of Figure 5 shows system times spent by this benchmark. This is useful to isolate the total effect on the CPU alone, since SCA-based file systems change data size and thus change the amount of disk I/O performed. Wrapfs adds 14.4% overhead over Ext2, because of the need to copy data pages between layers. Copyfs adds only 1.3% overhead over Wrapfs; this shows that our index file handling is fast. Compared to Copyfs, Uuencodefs adds 7% overhead and Gzipfs adds 69.9%. These are the costs of the respective SCAs in use and are unavoidable—whether running in the kernel or user-level.

The total size of an unencoded build of am-utils is 22.9MB; a Uuencoded build is one-third larger; Gzipfs reduces this size by a factor of 2.66 to 8.6MB. So while Uuencodefs increases disk I/O, it does not translate to a lot of additional system time because the Uuencode algorithm is trivial. Gzipfs, while decreasing disk I/O, however, is a costlier algorithm than Uuencode. That's why Gzipfs's system time overhead is greater overall than Uuencodefs's. The additional disk I/O performed by Copyfs is small and relative to the size of the index file.

The bottom part of Figure 5 shows elapsed times for this benchmark. These figures are the closest to what users will see in practice. Elapsed times factor in increased CPU times the more expensive the SCA is, as well as changes in I/O that a given file system performs: I/O for index file, increased I/O for Uuencodefs, and decreased I/O for Gzipfs.

On average, the cost of data copying without size-changing (Wrapfs compared to Ext2fs) is an additional

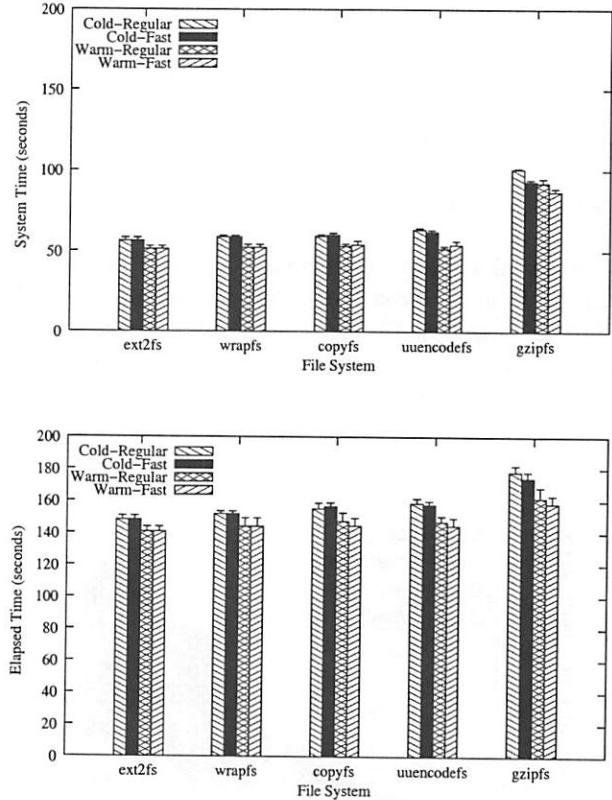


Figure 5: The Am-utils large-compile benchmark. Elapsed times shown on top and system times shown on bottom. The standard deviations for this benchmark were less than 3% of the mean.

2.4%. SCA support (Copyfs over Wrapfs) adds another 2.3% overhead. The Uuencode algorithm is simple and adds only 2.2% additional overhead over Copyfs. Gzipfs, however, uses a more expensive algorithm (Deflate) [7], and it adds 14.7% overhead over Copyfs. Note that the elapsed-time overhead for Gzipfs is smaller than its CPU overhead (almost 70%) because whereas the Deflate algorithm is expensive, Gzipfs is able to win back some of that overhead by its I/O savings.

Using a warm cache improves performance by 5–10%. Using fast tails improves performance by at most 2%. The code that is enabled by fast tails must check, for each read or write operation, if we are at the end of the file, if a fast tail already exists, and if a fast tail is large enough that it should be encoded and a new fast tail started. This code has a small overhead of its own. For file systems that do not need fast tails (e.g., Copyfs), fast tails add an overhead of 1%. We determined that fast tails is an option best used for expensive SCAs where many small appends are occurring, a conclusion demonstrated more visibly in Section 7.4.2.

7.3.2 Bonnie

Figure 6 shows the results of running Bonnie on the five file systems. Since Bonnie exercises data reading and writing heavily, we expect it to be affected by the SCA in use. This is confirmed in Figure 6. Over all runs in this benchmark, Wrapfs has an average overhead of 20% above Ext2fs, ranging from 2–73% for the given files. Copyfs only adds an additional 8% average overhead over Wrapfs. Uuencodefs adds an overhead over Copyfs that ranges from 5% to 73% for large files. Gzipfs, with its expensive SCA, adds an overhead over Copyfs that ranges from 22% to 418% on the large 128MB test file.

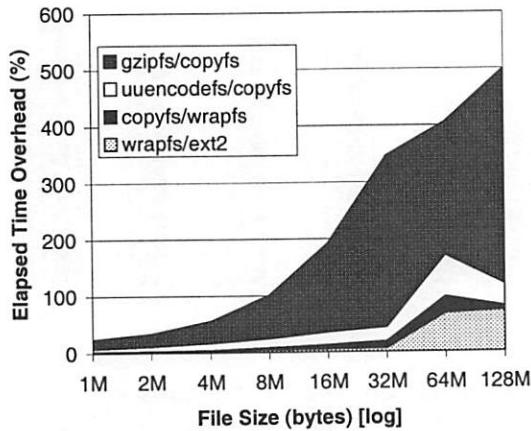


Figure 6: The Bonnie benchmark performs many repeated reads and writes on one file as well as numerous random seeks and writes in three concurrent processes. We show the total cumulative overhead of each file system. Note that the overhead bands for Gzipfs and Uuencodefs are each relative to Copyfs. We report the results for files 1MB and larger, where the overheads are more visible.

Figure 6 exhibits overhead spikes for 64MB files. Our test machines had 128MB of memory. Our stackable system caches two pages for each page of a file: one encoded page and one decoded page, effectively doubling the memory requirements. The 64MB files are the smallest test files that are large enough for the system to run out of memory. Linux keeps data pages cached for as long as possible. When it runs out of memory, Linux executes an expensive scan of the entire page cache and other in-kernel caches, purging as many memory objects as it can, possibly to disk. The overhead spikes in this figure occur at that time.

Bonnie shows that an expensive algorithm such as compression, coupled with many writes in the middle of large files, can degrade performance by as much as a factor of 5–6. In Section 9 we describe certain optimizations that we are exploring for this particular problem.

7.4 Micro-Benchmark Results

7.4.1 File-Copy

Figure 7 shows the results of running the file-copy benchmark on the different file systems. Wrapfs adds an average overhead of 16.4% over Ext2fs, which goes to 60% for a file size of 32MB; this is the overhead of data page copying. Copyfs adds an average overhead of 23.7% over Wrapfs; this is the overhead of updating and writing the index file as well as having to make temporary data copies (explained in Section 6) to support writes in the middle of files. The Uuencode algorithm adds an additional average overhead of 43.2% over Copyfs, and as much as 153% overhead for the large 32MB file. The linear overheads of Copyfs increase with the file's size due to the extra page copies that Copyfs must make, as explained in Section 6. For all copies over 4KB, fast-tails makes no difference at all. Below 4KB, it only improves performance by 1.6% for Uuencodefs. The reason for this is that this benchmark copies files only once, whereas fast-tails is intended to work better in situations with multiple small appends.

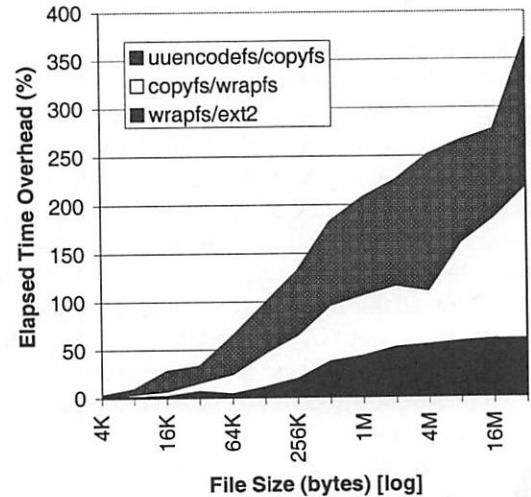


Figure 7: Copying files into a tested file system. As expected, Uuencodefs is costlier than Copyfs, Wrapfs, and Ext2fs. Fast-tails do not make a difference in this test, since we are not appending multiple times.

7.4.2 File-Append

Figure 8 shows the results of running the file-append benchmark on the different file systems. The figure shows the two emerging trends in effectiveness of the fast tails code. First, the more expensive the algorithm, the more helpful fast tails become. This can be seen in the right column of plots. Second, the smaller the number of bytes appended to the file is, the more savings fast tails provide, because the SCA is called fewer times. This can be seen as the

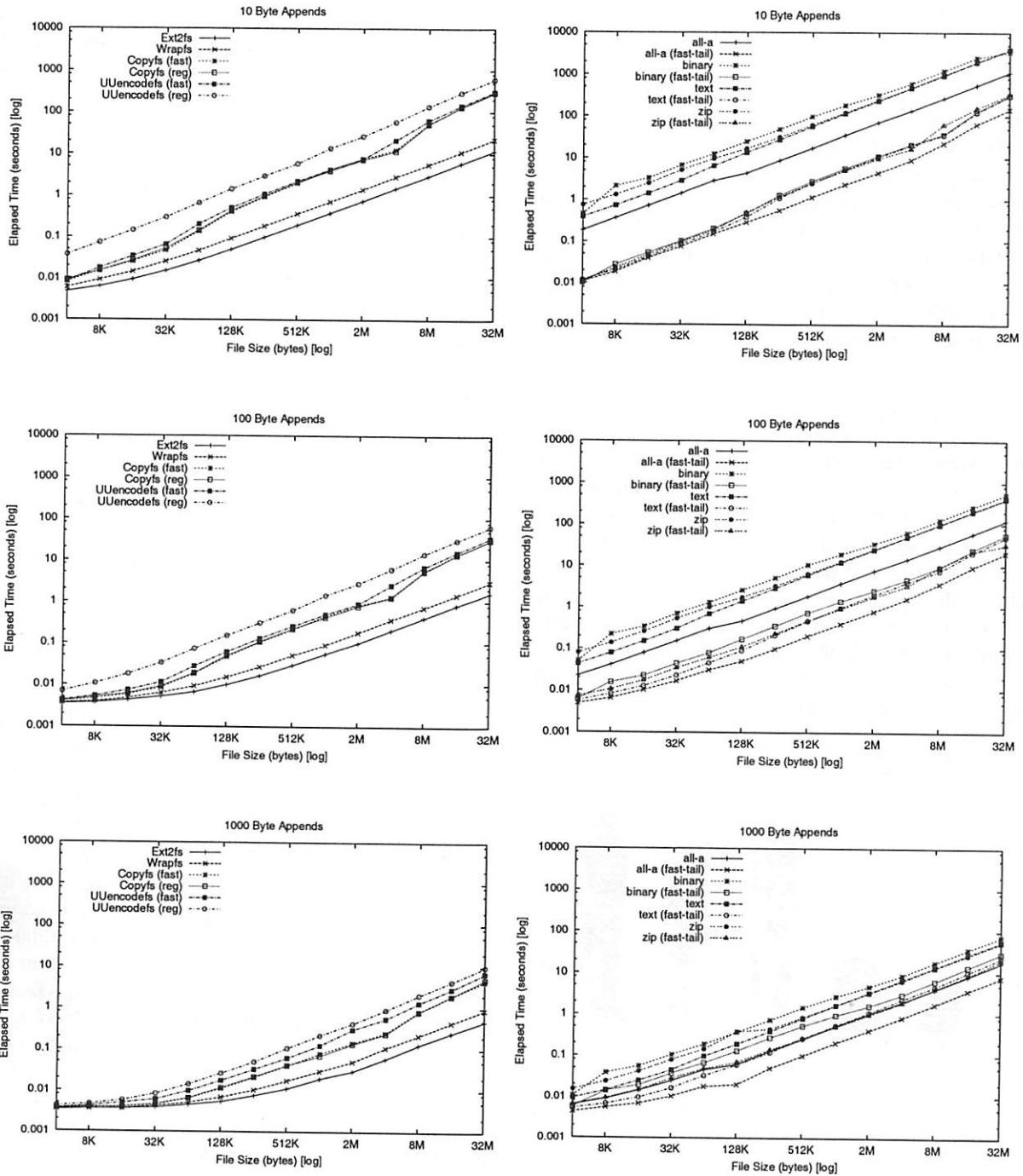


Figure 8: Appending to files. The left column of plots shows appends for Uuencodefs and Copyfs. The right column shows them for Gzipfs, which uses a more expensive algorithm; we ran Gzipfs on four different file types. The three rows of two plots each show, from top to bottom, appends of increasing sizes: 10, 100, and 1000 bytes, respectively. The more expensive the SCA is, and the smaller the number of bytes appended is, the more effective fast tails become; this can be seen as the trend from lower leftmost plot to the upper rightmost plot. The standard deviation for these plots did not exceed 9% of the mean.

trend from the bottom plots (1000 byte appends) to the top plots (10 byte appends). The upper rightmost plot clearly clusters together the benchmarks performed with fast tails support on and those benchmarks conducted without fast tails support.

Not surprisingly, there is little savings from fast tail support for Copyfs, no matter what the append size is. Uuencodefs is a simple algorithm that does not consume too much CPU cycles. That is why savings for using fast tails in Uuencodefs range from 22% for 1000-byte appends to a factor of 2.2 performance improvement for 10-byte appends. Gzipfs, using an expensive SCA, shows significant savings: from a minimum performance improvement factor of 3 for 1000-byte appends to as much as a factor of 77 speedup (both for moderately sized files).

7.4.3 File-Attributes

Figure 9 shows the results of running the file-attributes benchmark on the different file systems. Wraps add an overhead of 35% to the GETATTR file system operation because it has to copy the attributes from one inode data structure into another. SCA-based file systems add the most significant overhead, a factor of 2.6–2.9 over Wraps; that is because Copyfs, Uuencodefs, and Gzipfs include stackable SCA support, managing the index file in memory and on disk. The differences between the three SCA file systems in Figure 9 are small and within the error margin.

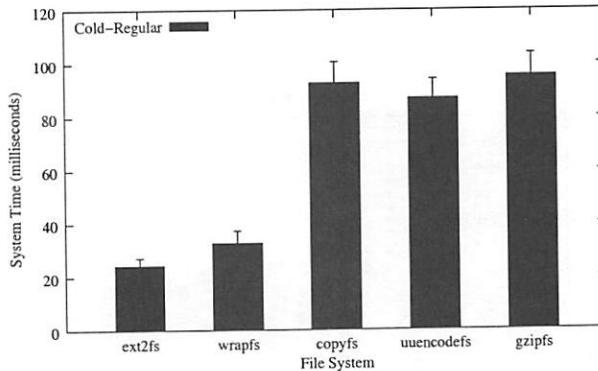


Figure 9: System times for retrieving file attributes using `lstat(2)` (cold cache)

While the GETATTR file operation is a popular one, it is still fast because the additional inode for the small index file is likely to be in the locality of the data file. Note that Figure 9 shows cold cache results, whereas most operating systems cache attributes once they are retrieved. Our measured speedup of cached vs. uncached attributes shows an improvement factor of 12–21. Finally, in a typical workload, bulk data reads and writes are likely to dominate any other file system operation such as GETATTR.

7.5 File System vs. User-Level Tool Results

Figure 10 shows the results of comparing Gzipfs against `gzip` using the file-copy benchmark. The reason Gzipfs is faster than `gzip` is primarily due to running in the kernel and reducing the number of context switches and kernel/user data copies.

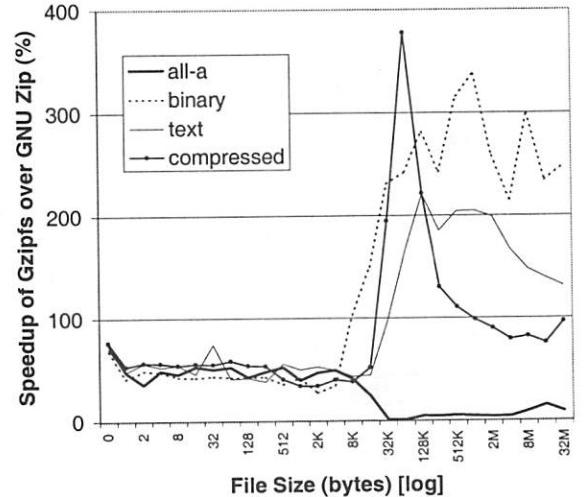


Figure 10: Comparing file copying into Gzipfs (kernel) and using `gzip` (user-level) for various file types and sizes. Here, a 100% speedup implies twice as fast.

As expected, the speedup for all files up to one page size is about the same, 43.3–53.3% on average; that is because the savings in context switches are almost constant. More interesting is what happens for files greater than 4KB. This depends on two factors: the number of pages that are copied and the type of data being compressed.

The Deflate compression algorithm is dynamic; it will scan ahead and back in the input data to try to compress more of it. Deflate will stop compressing if it thinks that it cannot do better. We see that for binary and text files, Gzipfs is 3–4 times faster than `gzip` for large files; this speedup is significant because these types of data compress well and thus more pages are manipulated at any given time by Deflate. For previously compressed data, we see that the savings is reduced to about double; that is because Deflate realizes that these bits do not compress easily and it stops trying to compress sooner (fewer pages are scanned forward). Interestingly, for the all-a file, the savings average only 12%. That is because the Deflate algorithm is quite efficient with that type of data: it does not need to scan the input backward and it continues to scan forward for longer. However, these forward-scanned pages are looked at few times, minimizing the number of data pages that `gzip` must copy between the user and the kernel. Finally, the plots in Figure 10 are not smooth because most of the input data is not uniform and thus it takes Deflate a different

amount of effort to compress different bytes sequences.

One additional benchmark of note is the space savings for Gzipfs as compared to the user level `gzip` tool. The Deflate algorithm used in both works best when it is given as much input data to work with at once. GNU zip looks ahead at 64KB of data, while Gzipfs currently limits itself to 4KB (one page). For this reason, `gzip` achieves on average better compression ratios: as little as 4% better for compressing previously compressed data, to 56% for compressing the all-a file.

We also compared the performance of Uuencodefs to the user level `uuencode` utility. Detailed results are not presented here due to space limitations, but we found the performance savings to be comparable to those with Gzipfs compared to `gzip`.

7.6 Additional Tests

We measured the time it takes to recover an index file and found it to be statistically indifferent from the cost of reading the whole file. This is expected because to recover the index file we have to decode the complete data file.

Finally, we checked the in-kernel memory consumption. As expected, the total number of pages cached in the page cache is the sum of the encoded and decoded files' sizes (in pages). This is because in the worst case, when all pages are warm and in the cache, the operating system may cache all encoded and decoded pages. For Copyfs, this means doubling the number of pages cached; for Gzipfs, fewer pages than double are cached because the encoded file size is smaller than the original file; for Uuencodefs, 2.33 times the number of original data pages are cached because the algorithm increased the data size by one-third. In practice, we did not find the memory consumption in stacking file systems on modern systems to be onerous [29].

8 Conclusions

The main contribution of our work is demonstrating that SCAs can be used effectively and transparently with stackable file systems. Our performance overhead is small and running these algorithms in the kernel improves performance considerably. File systems with support for SCAs can offer new services automatically and transparently to applications without having to change these applications or run them differently. Our templates provide support for generic SCAs, allowing developers to write new file systems easily.

Stackable file systems also offer portability across different file systems. File systems built with our SCA support can work on top of any other file system. In addition, we have done this work in the context of our FiST language, allowing rapid development of SCA-based file systems on multiple platforms [25, 29].

9 Future Work

We are investigating methods of improving the performance of writes in the middle of files by decoupling the order of the bytes in the encoded file from their order in the original file. By decoupling their order, we could move writes in the middle of files elsewhere—say the end of the file (similar to a journal) or an auxiliary file. Another alternative is to structure the file differently internally: instead of a sequential set of blocks, it could be organized as a B-tree or hash table where the complexity order of insertions in the middle is sub-linear. These methods would allow us to avoid having to shift bytes outward to make space for larger encoded units. However, if we begin storing many encoded chunks out of order, large files could get fragmented. We would need a method for compaction or coalescing all these chunks into a single sequential order.

An important optimization we plan to implement is to avoid extra copying of data into temporary buffers. This is only needed when an encoded buffer is written in the middle of a file and its encoded length is greater than its decoded length; in that case we must shift outward some data in the encoded data file to make room for the new encoded data. We can optimize this code and avoid making the temporary copies when files are appended to or being newly created and written sequentially.

10 Acknowledgments

We would like to thank Jerry B. Altzman for his initial input into the design of the index table. We thank John Heidemann for offering clarification regarding his previous work in the area of stackable filing. Thanks go to the Usenix reviewers and especially our shepherd, Margo Seltzer. This work was supported in part by NSF CISE Research Infrastructure grant EIA-9625374, an NSF CAREER award, and Sun Microsystems.

References

- [1] Am-utils (4.4BSD Automounter Utilities). Am-utils version 6.0.4 User Manual. February 2000. Available <http://www.cs.columbia.edu/~ezk/am-utils/>.
- [2] L. Ayers. E2compr: Transparent File Compression for Linux. *Linux Gazette*, Issue 18, June 1997. <http://www.linuxgazette.com/issue18/e2compr.html>.
- [3] M. Burrows, C. Jerian, B. Lampson, and T. Mann. Online data compression in a log-structured file system. *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 2–9.
- [4] M. I. Bushnell. The HURD: Towards a New Strategy of OS Design. *GNU's Bulletin*. Free Software Foundation, 1994. <http://www.gnu.org/software/hurd/hurd.html>.

- [5] V. Cate and T. Gross. Combining the concepts of compression and caching for a two-level filesystem. *Fourth ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA), pages 200-211.
- [6] R. Coker. The Bonnie Home Page. <http://www.textuality.com/bonnie>.
- [7] P. Deutsch. Deflate 1.3 Specification. RFC 1051. Network Working Group, May 1996.
- [8] J. L. Gailly. GNU zip. <http://www.gnu.org/software/gzip/gzip.html>.
- [9] J. L. Gailly and M. Adler. The zlib Home Page. <http://www.cdrom.com/pub/infozip/zlib/>.
- [10] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *Proceedings of the Summer USENIX Technical Conference*, pages 63–71, Summer 1990.
- [11] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Proceedings of Fifteenth ACM Symposium on Operating Systems Principles*. ACM SIGOPS, 1995.
- [12] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Tech-report CSD-910007. UCLA, 1991.
- [13] J. S. Heidemann and G. J. Popek. File System Development with Stackable Layers. *ACM ToCS*, 12(1):58–89, February 1994.
- [14] Y. A. Khalidi and M. N. Nelson. Extensible file systems in Spring. *Proceedings of Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–14, 1993.
- [15] S. R. Kleiman. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Proceedings of the Summer USENIX Technical Conference*, pages 238–47, Summer 1986.
- [16] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. *CompCon Conference Proceedings*, 1994.
- [17] L. Mummert and M. Satyanarayanan. Long Term Distributed File Reference Tracing: Implementation and Experience. Report CMU-CS-94-213. Carnegie Mellon University, Pittsburgh, U.S., 1994.
- [18] R. Nagar. Filter Drivers. In *Windows NT File System Internals: A developer's Guide*, pages 615–67. O'Reilly, 1997.
- [19] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. *Proceedings of Summer UKUUG Conference*, pages 1–9, July 1990.
- [20] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. *Proceedings of the Annual USENIX Technical Conference*, June 2000.
- [21] D. S. H. Rosenthal. Requirements for a “Stacking” Vnode/VFS Interface. UI document SD-01-02-N014. UNIX International, 1992.
- [22] D. S. H. Rosenthal. Evolving the Vnode Interface. *Proceedings of the Summer USENIX Technical Conference*, pages 107–18, Summer 1990.
- [23] B. Schneier. Algorithm Types and Modes. In *Applied Cryptography, 2nd ed.*, pages 189–97. John Wiley & Sons, 1996.
- [24] G. C. Skinner and T. K. Wong. “Stacking” Vnodes: A Progress Report. *Proceedings of the Summer USENIX Technical Conference*, pages 161–74, June 1993.
- [25] E. Zadok. *FiST: A System for Stackable File System Code Generation*. PhD thesis. Columbia University, May 2001.
- [26] E. Zadok and I. Bădulescu. A Stackable File System Interface for Linux. *LinuxExpo Conference Proceedings*, 1999.
- [27] E. Zadok, I. Bădulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CU-CS-021-98. Computer Science Department, Columbia University, 1998.
- [28] E. Zadok, I. Bădulescu, and A. Shender. Extending File Systems Using Stackable Templates. *Proceedings of the Annual USENIX Technical Conference*, June 1999.
- [29] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. *Proceedings of the Annual USENIX Technical Conference*, June 2000.

Software, documentation, and additional papers are available from <http://www.cs.columbia.edu/~ezk/research/fist/>.

Payload Caching: High-Speed Data Forwarding for Network Intermediaries

Ken Yocum and Jeff Chase
*Department of Computer Science
Duke University*
`{grant,chase}@cs.duke.edu *`

Abstract

Large-scale network services such as data delivery often incorporate new functions by interposing intermediaries on the network. Examples of forwarding intermediaries include firewalls, content routers, protocol converters, caching proxies, and multicast servers. With the move toward network storage, even static Web servers act as intermediaries to forward data from storage to clients.

This paper presents the design, implementation, and measured performance of *payload caching*, a technique for improving performance of host-based intermediaries. Our approach extends the functions of the network adapter to cache portions of the incoming packet stream, enabling the system to forward data directly from the cache. We prototyped payload caching in a programmable high-speed network adapter and a FreeBSD kernel. Experiments with TCP/IP traffic flows show that payload caching can improve forwarding performance by up to 60% in realistic scenarios.

1 Introduction

Data forwarding is increasingly common in large-scale network services. As network link speeds advance, networks are increasingly used to spread the functions of large servers across collections of networked systems, pushing functions such as storage into back-end networks. Moreover, systems for wide-area data delivery increasingly incorporate new functions — such as request routing, caching, and filtering — by “stacking” intermediaries in a pipeline fashion.

For example, a typical Web document may pass

*Author's address: Department of Computer Science, Duke University, Durham, NC 27708-0129 USA. This work is supported by the National Science Foundation (through EIA-9870724 and EIA-9972879), Intel Corporation, and Myricom.

through a series of forwarding steps along the path from its home on a file server to some client, passing through a Web server and one or more proxy caches. Other examples of forwarding intermediaries include firewalls, content routers, protocol converters [10], network address translators (NAT), and “overcast” multicast nodes [13]. New forwarding intermediaries are introduced in the network storage domain [14, 2], Web services [12], and other networked data delivery.

This paper investigates a technique called *payload caching* to improve data forwarding performance on intermediaries. In this paper, we define *forwarding* as the simple updating of packet headers and optional inspection of data as it flows through an intermediary. Note that data forwarding is more general than packet forwarding. While it encompasses host-based routers, it also extends to a wider range of these intermediary services.

Payload caching is supported primarily by an enhanced network interface controller (NIC) and its driver, with modest additional kernel support in the network buffering and virtual memory system. The approach is for the NIC to cache portions of the incoming packet stream, most importantly the packet data payloads (as opposed to headers) to be forwarded. The host and the NIC coordinate use of the NIC's payload cache to reduce data transfers across the I/O bus. The benefit may be sufficient to allow host-based intermediaries where custom architectures were previously required. Section 2 explains in detail the assumptions and context for payload caching.

This paper makes the following contributions:

- It explores the assumptions underlying payload caching, and the conditions under which it delivers benefits. Quantitative results illustrate the basic properties of a payload cache.

- It presents an architecture and prototype implementation for payload caching in a programmable high-speed network interface, with extensions to a zero-copy networking framework [5] in a FreeBSD Unix kernel. This design shows how the host can manage the NIC's payload cache for maximum flexibility.
- It presents experimental results from the prototype showing forwarding performance under payload caching for a range of TCP/IP networking traffic. The TCP congestion control scheme adapts to deliver peak bandwidth from payload caching intermediaries.
- It outlines and evaluates an extension to payload caching, called *direct forwarding*, that improves forwarding performance further when intermediaries access only the protocol headers.

This paper is organized as follows. Section 2 gives an overview of payload caching and its assumptions. Section 3 outlines interfaces and extensions for payload caching at the boundary between a host and its NIC. Section 4 describes our payload caching prototype using Myrinet and FreeBSD. Section 5 examines the behavior and performance of payload caching. Section 6 describes related work and outlines future research. Section 7 concludes.

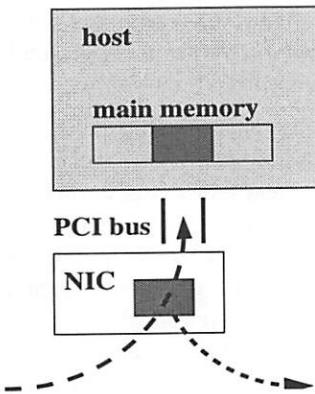


Figure 1: Forwarding a data payload with payload caching.

2 Overview

The payload caching technique optimizes network communication for forwarding intermediaries. Payload caching targets a typical host-based structure, in which the forwarding logic runs on a CPU whose memory is separated from the network interface.

The NIC moves data to and from host memory using Direct Memory Access (DMA) across an I/O bus, such as PCI.

A forwarding intermediary receives a stream of packets from the network. Each packet DMAs across the I/O bus into one or more buffers in host memory. The network protocol stack inspects the headers and delivers the data to an application containing the intermediary logic, such as a firewall or caching proxy. The application may examine some of the data, and it may forward some or all of the data (the payload) to another destination without modifying it.

Figure 1 shows the potential benefit of payload caching in this scenario. Ordinarily, forwarded data payloads cross the I/O bus twice, once on input and once on output. Payload caching leaves incoming payloads in place in NIC buffers after delivering them to the host. If the host forwards the data unchanged, and if the forwarded data is still cached on the NIC, then the output transfer across the bus is unnecessary. This reduces the bandwidth demand of forwarding on the I/O bus and memory system, freeing these resources for other I/O or memory-intensive CPU activity. Payload caching can be especially effective for intermediaries that do I/O to other devices, such as disk-based Web proxy caches.

Payload caching imposes little or no runtime cost, but it yields a significant benefit under the following conditions.

- The intermediary forwards a large share of its incoming data without modifying it. This is often the case for intermediaries for Web delivery, including caching proxies, firewalls, content routers, multicast overlay nodes, and Web servers backed by network storage. Payload caching also naturally optimizes multicast transmits, such as mirrored writes to a network storage server or to a network memory cache [9].
- The payload cache on the NIC is large enough to retain incoming payloads in the cache until the host can process and forward them. In practice, the amount of buffering required depends on the incoming traffic rate, traffic burstiness, and the CPU cost to process forwarded data. One contribution of this work is to empirically determine the hit rates for various payload cache sizes for TCP/IP streams. Section 5.3 presents experimental results that show good hit rates at forwarding speeds up to

1 Gb/s and payload cache sizes up to 1.4 MB.

- Forwarded data exits the intermediary by the same network adapter that it arrived on. This allows the adapter to obtain the transmitted data from its payload cache instead of from the intermediary’s memory. Note that this does not require that the output link is the same as the input link, since many recent networking products serve multiple links from the same adapter for redundancy or higher aggregate bandwidths. Payload caching provides a further motivation for multi-ported network adapters.
- The NIC supports the payload cache buffering policies and host interface outlined in Section 3. Our prototype uses a programmable Myrinet NIC, but the scheme generalizes easily to a full range of devices including Ethernet and VI NICs with sufficient memory.

While the payload caching idea is simple and intuitive, it introduces a number of issues for its design, implementation, and performance. How large must a payload cache be before it is effective? What is the division of function between the host and the NIC for managing a payload cache? How does payload caching affect other aspects of the networking subsystem? How does payload caching behave under the networking protocols and scenarios used in practice? The rest of this paper addresses these questions.

3 Design of Payload Caching

This section outlines the interface between the host and the NIC for payload caching, and its role in the flow of data through the networking subsystem.

The payload cache indexes a set of buffers residing in NIC memory. The NIC uses these memory buffers to stage data transfers between host memory and the network link. For example, the NIC handles an incoming packet by reading it from the network link into an internal buffer, then using DMA to transmit the packet to a buffer in host memory. All NICs have sufficient internal buffer memory to stage transfers; payload caching requires that the NIC contain sufficient buffer memory to also serve as a cache. For simplicity, this section supposes that each packet is cached in its entirety in a single host buffer and a single NIC buffer, and that the payload cache is fully effective even if the host forwards only portions of each packet unmodified. Section 4 fills

in important details of host and NIC buffering left unspecified in this section.

The host and NIC coordinate use of the payload cache and cooperate to manage associations between payload cache entries and host buffers. A key goal of our design is to allow the host — rather than the NIC — to efficiently manage the placement and eviction in the NIC’s payload cache. This simplifies the NIC and allows flexibility in caching policy for the host.

Figure 2 depicts the flow of buffer states and control through the host’s networking subsystem. Figure 3 gives the corresponding state transitions for the payload cache. The rest of this section refers to these two figures to explain interactions between the host and the NIC for payload caching.

The dark horizontal bar at the top of Figure 2 represents the boundary between the NIC and the host. We are concerned with four basic operations that cross this boundary in a typical host/NIC interface. The host initiates *transmit* and *post receive* operations to send or receive packets. For example, the host network driver posts a receive by appending an operation descriptor to a NIC receive queue, specifying a host buffer to receive the data; the NIC delivers an incoming packet header and payload by initiating a DMA operation from NIC memory to the host buffer. In general, there are many outstanding receives at any given time, as the host driver attempts to provide the NIC with an adequate supply of host buffers to receive the incoming packet stream. When a transmit or receive operation completes, the NIC signals *receive* and *transmit complete* events to the host, to inform it that the NIC is finished filling or draining buffers for incoming or outgoing packets.

Payload caching extends these basic interactions to enable the host to name NIC buffers in its commands to the NIC. This allows the host to directly control the payload cache and to track NIC buffers that have valid cached images of host buffers. To avoid confusion between host memory buffers and internal NIC buffers, we refer to NIC buffers as payload cache *entries*. For the remainder of this paper, any use of the term *buffer* refers to a host memory buffer, unless otherwise specified.

Each payload cache entry is uniquely named by an *entry ID*. The host network driver specifies an entry ID of a NIC buffer to use for each host buffer in a newly posted transmit or receive. This allows the host to control which internal NIC buffers are used to stage transfers between host memory and the net-

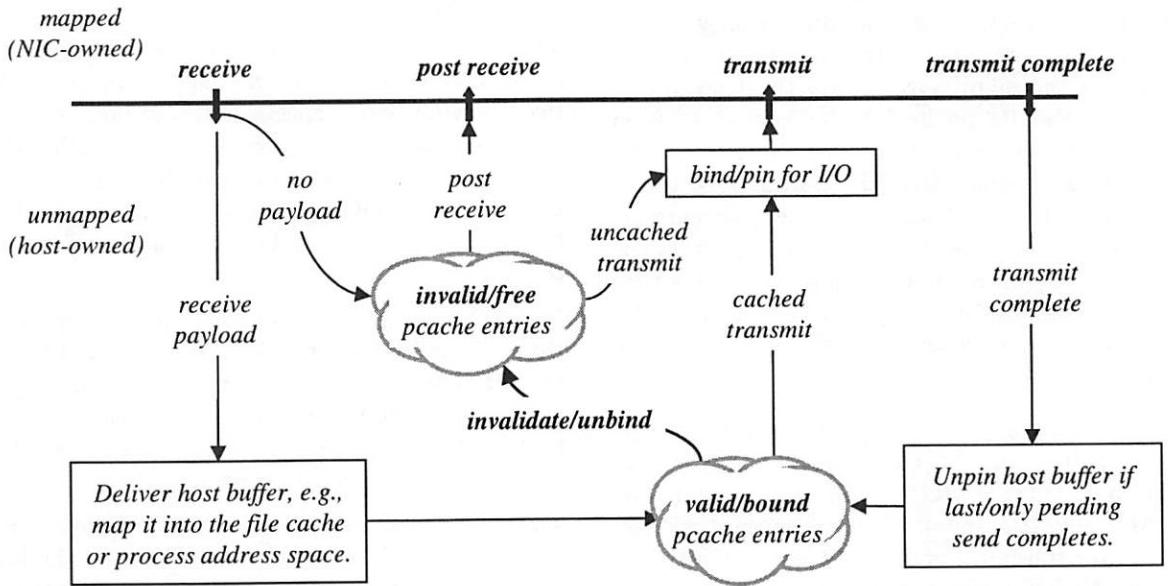


Figure 2: The flow of host buffers and payload cache entries through the networking subsystem.

work links. The NIC retains the data from each transfer in the corresponding entry until the host commands the NIC to reuse that entry for a subsequent transfer. Thus each transfer effectively loads new data into the payload cache; the host maintains an association between the host buffer and its payload cache entry as long as the entry's cached image of the buffer remains valid. If the host then initiates a subsequent transmit from the same buffer without modifying the data, the host sets a field in the descriptor informing the NIC that it may transmit data cached in the specified entry rather than fetching the data from the host buffer using DMA. This is a payload cache hit.

By specifying the entry ID for a transmit or receive, the host also controls eviction of data from the payload cache. This is because I/O through a payload cache entry may displace any data previously cached in the target entry. It is easy to see that most-recently-used (MRU) is the best replacement policy for the payload cache when the host forwards data in FIFO order. This is discussed further in Section 5.3.

We use the following terminology for the states of payload cache entries and host buffers. An entry is *valid* if it holds a correct copy of some host buffer, else it is *invalid*. A host buffer is *cached* if some valid entry holds a copy of it in the payload cache, else it is *uncached*. An entry is *bound* if it is associated with a buffer, else it is *free*. A buffer is *bound* if it is associated with an entry, else it is *unbound*.

A bound (*buffer,entry*) pair is *pending* if the host has posted a transmit or receive operation to the NIC specifying that pair and the operation has not yet completed. Note that a bound buffer may be uncached if it is pending.

Initially, all entries are in the *free* state. The host driver maintains a pool of entry IDs for free payload cache entries, depicted by the cloud near the center of Figure 2. The driver draws from this pool of free entries to post new receives, and new transmits of uncached buffers. Before initiating the I/O, the operating system pins its buffers, binds them to the selected payload cache entries, and transitions the entries to the *pending* state. When the I/O completes, the NIC notifies the host with a corresponding *receive* or *transmit complete* notification via an interrupt. A *receive* may complete without depositing valid cacheable data into some buffer (e.g., if it is a short packet); in this case, the driver immediately unbinds the entry and returns it to the free pool. Otherwise, the operating system delivers the received data to the application and adds the bound (*buffer,entry*) pair to its *bound entry* pool, represented by the cloud in the lower right of Figure 2.

On a transmit, the driver considers whether each buffer holding the data to be transmitted is bound to a valid payload cache entry. If the buffer is unbound, the driver selects a new payload cache entry from the free pool to stage the transfer from the buffer. If the buffer is already bound, this indicates

that the host is transmitting from the same buffer used in a previous transmit or receive, e.g., to forward the payload data to another destination. This yields a payload cache hit if the associated entry is valid. The host reuses the payload cache entry for the transmit, and sets a field in the operation descriptor indicating that the entry is still valid.

After the transmit completes, the driver adds the entry and buffer pairing to the bound entry pool. Regardless of whether the transmit was a payload cache hit, the entry is now valid and bound to the host buffer used in the transmit. A subsequent transmit of the same data from the same buffer (e.g., as in a multicast) yields a payload cache hit.

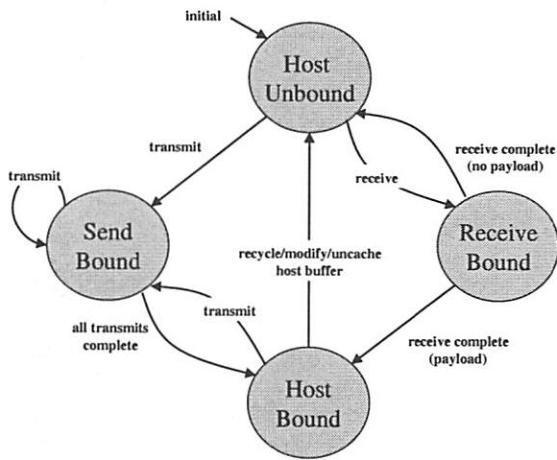


Figure 3: Payload cache entry states and transitions.

Figure 3 summarizes the states and transitions for payload cache entries. Initially, all entries are in the free state at the top of the figure. If the driver posts a transmit or a receive on an unbound/uncached host buffer, it selects a free NIC payload cache entry to bind to the buffer and stage the transfer between the network link and host memory. This causes the selected entry to transition to the left-hand **send-bound** state for a pending transmit, or to the right-hand **receive-bound** state for a pending receive.

In the **send-bound** and **receive-bound** states in Figure 3, the entry and buffer are bound with a pending I/O operation. For a transmit, the entry is marked valid as soon as the transfer initiates; this allows subsequent transmits from the same buffer (e.g., for a multicast) to hit in the payload cache, but it assumes that the NIC processes pending transmits in FIFO order. For a receive, the entry is marked valid only on completion of the received

packet, and only if the received packet deposited cacheable data in the posted buffer (a short packet might not occupy all posted buffers).

A valid payload cache entry transitions to the bottom **host-bound** state when the pending transmit or receive completes. In this state, the entry retains its association with the host buffer, and caches a valid image of the buffer left by the completed I/O. Subsequent transmits from the buffer in this state lead back to **send-bound**, yielding a payload cache hit.

Once a binding is established between a host buffer and a valid payload cache entry (the **host-bound** state in Figure 3, and the bottom cloud in Figure 2), the operating system may break the binding and invalidate the payload cache entry. This returns the payload cache entry to the free pool, corresponding to the initial **host-unbound** state in Figure 3, or to the top cloud in Figure 2. This system must take this transition in the following cases:

- The system delivers the payload data to some application, which subsequently modifies the data, invalidating the associated payload cache entry.
- The system links the data buffer into the system file cache, and a process subsequently modifies it, e.g., using a *write* system call.
- The system releases the buffer and recycles the memory for some other purpose.
- The system determines that the cached entry is not useful, e.g., it does not intend to forward the data.
- There are no free payload cache entries, and the driver must evict a bound entry in order to post a new transmit or receive operation.

The payload cache module exports an interface to higher levels of the OS kernel to release or invalidate a cache entry for these cases. In all other respects payload caching is hidden in the NIC driver and is transparent to upper layers of the operating system.

4 Implementation

This section describes a prototype implementation of payload caching using Myrinet, a programmable high-speed network interface. It extends the design overview in the previous section with details relating to the operating system buffering policies.

Payload Cache Operations (exported to network driver)	
pc_receive_bind(physaddr)	Invalidate old binding if present, and bind the replacement payload cache entry with a host physical frame.
pc_send_bind(physaddr)	If the buffer is cached on the adapter, use existing binding, else find replacement and create new binding.
pc_receive_complete(physaddr)	The cache entry is now valid/bound.
pc_send_complete(physaddr)	If this is the last outstanding send, the cache entry is now valid/bound.
Payload Cache Management (exported to operating system)	
pc_invalidate_binding(physaddr)	Invalidate the payload cache entry bound to this physical address; the entry is now <i>host_unbound</i> .
pc_advise(physaddr, options)	Advise the payload cache manager to increase or decrease the payload cache entries priority.

Table 1: Payload Cache module APIs for the network driver and OS kernel.

We implemented payload caching as an extension to Trapeze [1, 4], a firmware program for Myrinet, and associated Trapeze driver software for the FreeBSD operating system. The host-side payload cache module is implemented by 1600 lines of new code alongside a Trapeze device support package below the driver itself. While our prototype implementation is Myrinet-specific, the payload caching idea applies to Gigabit Ethernet and other network interfaces.

Our prototype integrates payload caching with FreeBSD extensions for zero-copy TCP/IP networking [5]. This system uses page remapping to move the data between applications and the operating system kernel through the socket interface, avoiding data copying in many common cases. This allows us to explore the benefit of payload caching for intermediaries whose performance is not dominated by superfluous copying overhead. Copy avoidance also simplifies the payload cache implementation because forwarded data is transmitted from the same physical host buffer used to receive it. Thus there is at most one host buffer bound to each payload cache entry.

The Trapeze network interface supports page remapping for TCP/IP networking by separating protocol headers from data payloads, and depositing payloads in page-aligned host payload buffers allocated from a pool of VM page frames by the driver. The payload caching prototype manages a simple one-to-one mapping of bound payload cache entries with cached host memory page frames; the buffer bound to each payload cache entry is identified by a simple physical address.

Any modification to a cached buffer page in the host invalidates the associated payload cache entry, if any. Page protections may be used to trap buffer

updates in user space. Note, however, that changes or reconstruction of packet headers does not invalidate the cache entries for the packet payload. For example, a Web server accessing files from an NFS file server and sending them out over an HTTP connection may use the payload cache effectively.

4.1 Payload Cache Module

Table 1 shows the interface exported by the payload cache module (*pcache*) to the Trapeze network driver and upper kernel layers. When the driver posts a transmit or receive, it invokes the *pc_receive_bind* or *pc_send_bind* routine in *pcache* to check the binding state of the target host buffer frames, and establish bindings to payload cache entries if necessary. The *pcache* module maintains a *pcache* entry table storing the physical address of the buffer bound to each entry, if any, and a *binding table* storing an entry ID for each frame of host memory. If a posted buffer frame is not yet bound to an entry, *pcache* finds a free entry or a suitable bound entry to evict.

When a send or receive completes, the driver invokes the *pcache pc_send_complete* or *pc_receive_complete* routine. If there are no more pending I/O operations on an entry, and the recently completed I/O left the entry with valid data, then *pcache* transitions the entry to the **host-bound** state, shown earlier in Figure 3.

The *pcache* module exports routines called *pc.invalidate_binding* and *pc.advise* to the upper layers of the operating system kernel. The kernel uses these to invalidate a payload cache entry when its bound buffer is modified, or to inform *pcache* that the cached data is or is not valuable. For example, the OS may call *pc.advise* to mark an entry as an eviction candidate if its payload will

not be forwarded.

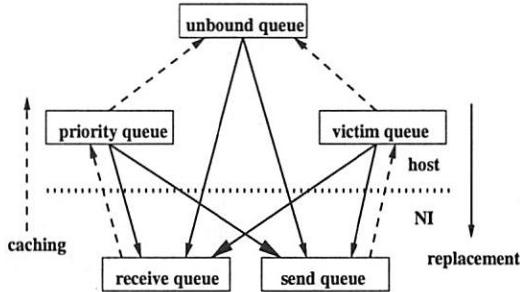


Figure 4: Queues for payload caching and replacement.

The prototype gives eviction priority to data that has been received but not yet transmitted. Any payload cache entry that is not pending resides on one of three replacement queues: unbound (free), priority, and victim. Figure 4 shows the movement of payload cache entries between these queues and the NIC send/receive queues. Entries for completed sends move to the victim queue, while entries for completed receives move to the priority queue. Entries on either victim or priority transition to the unbound queue if they are invalidated or demoted by *pc_advise*. An evicted entry may come from any of these queues, in the following preference order: unbound, victim, priority.

Note that *pcache* manages the payload cache entirely within the host, including invalidation and replacement. Support for payload caching on the NIC is trivial. The host piggybacks all payload cache directives on other commands to the NIC (transmit and post receive), so payload caching imposes no measurable device or I/O overhead.

4.2 Direct Forwarding

In normal forwarding operation, a payload caching host receives control and payload from the NIC, but transmits only headers across the I/O bus, sending forwarded payloads from the cache. For intermediaries that do not access most payloads — such as protocol translators, multicast nodes, or content switches — a natural progression is to extend the separation of control and payload data paths. In this case the NIC only passes control headers to the host, not data payloads. We term this configuration DIRECT forwarding (in contrast to PCACHE forwarding). Our prototype supports DIRECT forwarding mode with a small extension to the NIC firmware and a small change to the *pcache* module and driver. Payload cache entry management does not change.

Experimental results in Section 5 show that DIRECT enables forwarding at link speeds, limited only by the CPU overhead for the forwarding logic. However, a pure DIRECT policy is appropriate only when the payload cache is adequately sized for the link or if the send rate is held below the level that overflows the cache. This is because evictions in a DIRECT payload cache discard the packet data, forcing the driver to drop any packet that misses in the payload cache in DIRECT mode.

Section 5.5 shows that TCP congestion control adapts to automatically deliver maximum allowable bandwidth through a DIRECT forwarder with very low miss rates in the presence of these packet drops. Even so, DIRECT is narrowly useful as implemented in our prototype. It would be possible to enhance its generality by extending the NIC to DMA direct-cached payloads to the host before eviction or on demand. Another alternative might be to extend the NIC to adaptively revert from DIRECT to PCACHE as it comes under load. We have not implemented these extensions in our prototype, but our implementation is sufficient to show the potential performance benefit from these more general approaches.

5 Payload Caching Performance

This section explores the effectiveness of the payload caching prototype for a simple kernel-based forwarding proxy. The results show the effect of payload caching on forwarding latency and bandwidth for TCP streams and UDP packet flows, varying the payload cache size, number of concurrent streams, packet size, and per-packet processing costs in the forwarding host CPU.

While the hit rate in the payload cache directly affects the increase in throughput and decrease in latency, it is not simply a function of cache size or replacement policy. Understanding the interplay between payload caching and forwarder behavior allows us to establish “real-world” performance under a variety of scenarios.

5.1 Experimental Setup

We ran all experiments using Dell PowerEdge 4400 systems on a Trapeze/Myrinet network. The Dell 4400 has a 733 MHz Intel Xeon CPU (32KB L1 cache, 256KB L2 cache), a ServerWorks ServerSet III LE chipset, and 2-way interleaved RAM. End systems use M2M-PCI64B Myrinet adapters with 66 MHz LANai-7 processors. The forwarder uses a more powerful pre-production prototype Myrinet 2000 NIC with a 132 MHz LANai-9 processor, which

Packet Size	Point to Point	Forwarding	PCACHE	DIRECT
1.5 KB	82.31 μ s	153.86 μ s	140.15 μ s	131.5 μ s
4 KB	108.36 μ s	224.68 μ s	191.68 μ s	173.71 μ s
8 KB	159.2 μ s	326.88 μ s	285.94 μ s	260.74 μ s

Table 2: One-way latency of UDP packets through an intermediary.

does not saturate at the forwarding bandwidths achieved with payload caching. The Myrinet 2000 NIC on the forwarder uses up to 1.4 MB of its onboard RAM as a payload cache in our experiments. All NICs, running Trapeze firmware enhanced for payload caching, are connected to PCI slots matched to the 1 Gb/s network speed. Since the links are bidirectional, the bus may constrain forwarding bandwidth.

All nodes run FreeBSD 4.0 kernels. The forwarding proxy software used in these experiments consists of a set of extensions to an IP firewall module in the FreeBSD network stack. The forwarder intercepts TCP traffic to a designated virtual IP address and port, and queues it for a kernel thread that relays the traffic for each connection to a selected end node. Note that the forwarder acts as an intermediary for the TCP connection between the end nodes, rather than maintaining separate connections to each end node. In particular, the forwarder does no high-level protocol processing for TCP or UDP other than basic header recognition and header rewriting to hide the identity of the endpoints from each other using Network Address Translation (NAT). This software provides a basic forwarding mechanism for an efficient host-based content switch or load-balancing cluster front end. It is equivalent to the kernel-based forwarding supported for application-level proxies by TCP splicing [8].

To generate network traffic through the forwarder we used *netperf* version 2.1pl3, a standard tool for benchmarking TCP/IP networking performance, and *Flowgen*, a network traffic generator from the DiRT project at UNC.

5.2 Latency

Table 2 gives the latency for one-way UDP transfers with packet sizes of 1500 bytes, 4KB, and 8KB. Interposing a forwarding intermediary imposes latency penalties ranging from 86% for 1500-byte packets to 105% for 8KB packets. Payload caching (PCACHE) reduces this latency penalty modestly, reducing forwarding latency by 8% for 1500-byte packets, 14% for 4KB packets, and 12% for 8KB pack-

ets. Direct forwarding (DIRECT) reduces forwarding latency further: the total latency improvement for DIRECT is 14% for 1500-byte packets, 22% for 4KB packets, and 20% for 8KB packets.

This experiment yields a payload cache hit for every forwarded packet, regardless of cache size. The resulting latency savings stems from reduced I/O bus crossings in the forwarder. PCACHE eliminates the I/O bus crossing on transmit, and DIRECT eliminates bus crossings on both transmit and receive. For all experiments the NIC uses a store-and-forward buffering policy, so I/O bus latencies are additive.

Propagation delays are higher in wide-area networks, so the relative latency penalty of a forwarding intermediary is lower. Therefore, the relative latency benefit from payload caching is also lower.

5.3 Payload Cache Size and Hit Rate

The next experiment explores the role of the forwarder's packet processing overhead on payload cache hit rates across a range of cache sizes. It yields insight into the amount of NIC memory needed to achieve good payload cache hit rates under various conditions.

The NIC deposits incoming packets into host memory as fast as the I/O bus and NIC resources allow, generating interrupts to notify the host CPU of packet arrival. In FreeBSD, the NIC driver's receiver interrupt handler directly invokes the IP input processing routine for all new incoming packets; this runs the protocol and places the received data on an input queue for delivery to an application. Incoming packets may accumulate on these input queues if the forwarder CPU cannot keep up with the incoming traffic, or if the incoming traffic is bursty. This is because the NIC may interrupt the application for service from the driver as more packets arrive.

The behavior of these queues largely determines the hit rates in the payload cache. Consider our simple forwarder example. The forwarder application runs as a thread within the kernel, and the network driver's incoming packet handler may interrupt it.

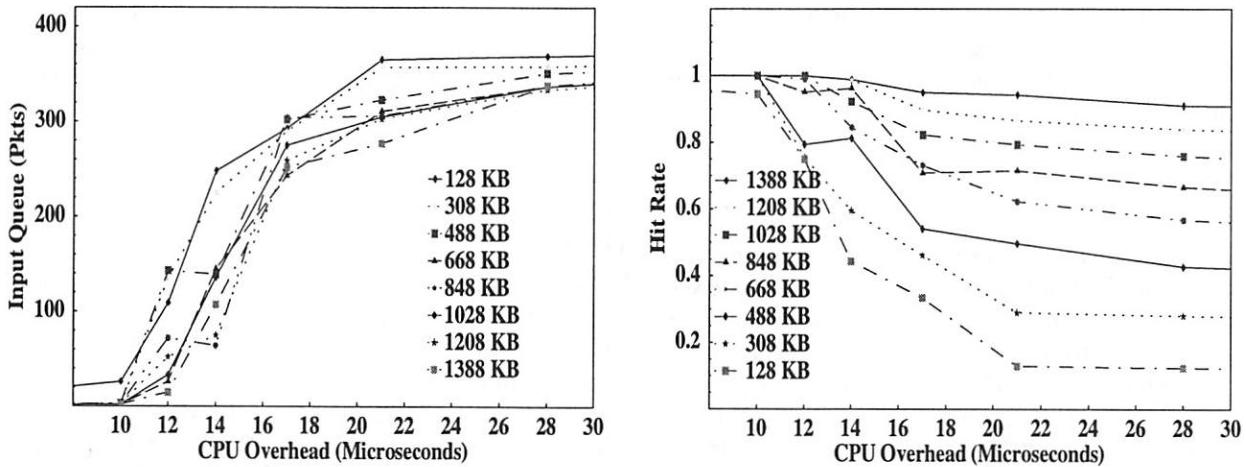


Figure 5: Forwarder queue size and hit rate for a 25 MB/s (200 Mb/s) stream of 4KB UDP packets. Each line shows results for a given effective payload cache size across a range of per-packet forwarding overheads on the *x*-axis.

As the driver consumes each incoming packet accepted by the NIC, it allocates a new host buffer and payload cache entry to post a new receive so the NIC may accept another incoming packet. Under ideal conditions the driver simply allocates from unused entries released as the forwarder consumes packets from its input queue and forwards their payloads. However, suppose traffic bursts or processing delays cause packets to accumulate on the input queue, awaiting forwarding. Since each buffered packet consumes space in the payload cache, this forces *pcache* to replace payload cache entries for packets that have not yet been forwarded, reducing the hit rate.

It is easy to see that under MRU replacement the payload cache hit rate for the forwarder will roughly equal the percentage of the buffered packet payloads that fit in the payload cache. Once an MRU cache is full, any new entries are immediately evicted. With FIFO forwarding, every buffered payload that found space in the cache on arrival ultimately yields one hit when it is forwarded; every other buffered payload ultimately yields one miss. Thus the average hit rate can be found by determining the average forwarder queue length — the number of payloads buffered for forwarding in the host — and dividing into the payload cache size. Note that MRU is the optimal behavior in this case because there can be no benefit to replacing an older cached payload with a newer one until the older one has been forwarded.

Queuing theory predicts these forwarder queue lengths under common conditions, as a function of

forwarder CPU overhead (per-packet CPU service demand) or CPU utilization. This experiment illustrates this behavior empirically, and also shows an interesting feedback effect of payload caching. We added a configurable per-packet CPU demand to the forwarder thread, and measured forwarder queue lengths and payload cache hit rate under a 25 MB/s (200 Mb/s) load of 4KB UDP packets. We used the UNC *Flowgen* tool to generate poisson-distributed interarrival gaps for even queue behavior. This allows us to explore the basic relationship between CPU demand and payload cache hit rate without the “noise” of the burstier packet arrivals common in practice.

The left-hand graph of Figure 5 shows the average number of packets queued in the intermediary as a function of the CPU demand. We ran several experiments varying the *effective payload cache size*, the number of payload cache entries not reserved for pending receives. As expected, the queues grow rapidly as the CPU approaches saturation. In this case, the OS kernel bounds the input packet queue length at 400 packets; beyond this point the IP input routine drops incoming packets at the input queue. This figure also shows that the queues grow more slowly for large payload cache sizes. Queuing theory also predicts this effect: hits in the payload cache reduce the effective service demand for each packet by freeing up cycles in the I/O bus and memory system.

The right-hand graph of Figure 5 shows the average payload cache hit rate for the same runs. At

low service demands, all packets buffered in the forwarder fit in the payload cache, yielding hit rates near 100%. As the forwarder queue lengths increase, a smaller share of the packet queue fits in the payload cache, and hit rates fall rapidly. As expected, the hit rate for each experiment is approximated by the portion of the packet queue that fits in the payload cache.

This experiment shows that a megabyte-range payload cache yields good hit rates if the CPU is powerful enough to handle its forwarding load without approaching saturation. As it turns out, this property is independent of bandwidth; if CPU power scales with link rates then payload caching will yield similar hit rates at much higher bandwidths. On the other hand, payload caching is not effective if the CPU runs close to its capacity, but this case is undesirable anyway because queueing delays in the intermediary impose high latency.

5.4 UDP Forwarding Bandwidth

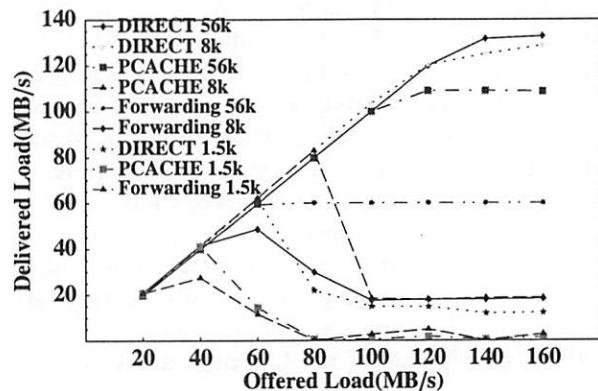


Figure 6: UDP forwarding bandwidth with PCACHE and DIRECT, for MTUs of 56KB, 8KB, and 1500 bytes.

The next experiment shows the bandwidth benefits of payload caching for *netperf* UDP packet flows. Figure 6 shows delivered UDP forwarding bandwidth as a function of input traffic rate for packet sizes of 1500 bytes, 8KB, and 56KB. The payload cache size is fixed at 1.4 MB. Bandwidth is measured as the number of bytes arriving at the receiver per unit of time. For these experiments, smaller packet sizes saturate the CPU at modest bandwidths; since UDP has no congestion control, the saturated intermediary drops many packets off of its input queues, but only after consuming resources to accept them. This livelock causes a significant drop in delivered bandwidth beyond saturation.

With 1500-byte packets, packet handling costs quickly saturate the forwarder's CPU, limiting forwarding bandwidth to 29 MB/s. PCACHE improves peak forwarding bandwidth by 35% to 40 MB/s. In this case, the benefit from PCACHE stems primarily from reduced memory bandwidth demand to forward each packet, as hits in the payload cache reduce the number of bytes transmitted from host memory.

The 8KB packets reduce packet handling costs per byte of data transferred, easing the load on the CPU. In this case, the CPU and the I/O bus are roughly in balance, and both are close to saturation at the peak forwarding rate of 48 MB/s. PCACHE improves the peak forwarding bandwidth by 75% to 84 MB/s due to reduced load on the I/O bus and on the forwarder's memory.

With 56KB packets, forwarding performance is limited only by the I/O bus. The base forwarding rate is roughly one-half the I/O bus bandwidth at 60 MB/s, since each payload crosses the bus twice. With PCACHE, the forwarding bandwidth doubles to 110 MB/s, approaching the full I/O bus bandwidth. This shows that payload caching yields the largest benefit when the I/O bus is the bottleneck resource, since it cuts bus utilization by half under ideal conditions. A faster CPU would show a similar result at smaller packet sizes. Note that for 56KB packets the forwarding rate never falls from its peak. This is because the CPU is not saturated; since the I/O bus is the bottleneck resource, the input rate at the forwarder is limited by link-level flow control on the Myrinet network. This is the only significant respect in which our experiments are not representative of more typical IP networking technologies.

The DIRECT policy reduces memory and I/O bus bandwidth demands further, and sustains much higher bandwidth across all packet sizes. At 1500 bytes, DIRECT reduces CPU load further than PCACHE, yielding a peak forwarding bandwidth of 60 MB/s. DIRECT can forward at a full 1 Gb/s for a 56KB packet size, with the CPU load at 20% and the I/O bus mostly idle for use by other devices. However, a payload cache miss affects bandwidth much more for DIRECT than for PCACHE, since a miss results in a packet drop. In these experiments, DIRECT suffered a 22% miss rate for a 100 MB/s input rate with 8KB MTU. The next section shows that TCP congestion control adapts to throttle the send rate when packets are dropped, yielding the best bandwidth and high hit rates for DIRECT.

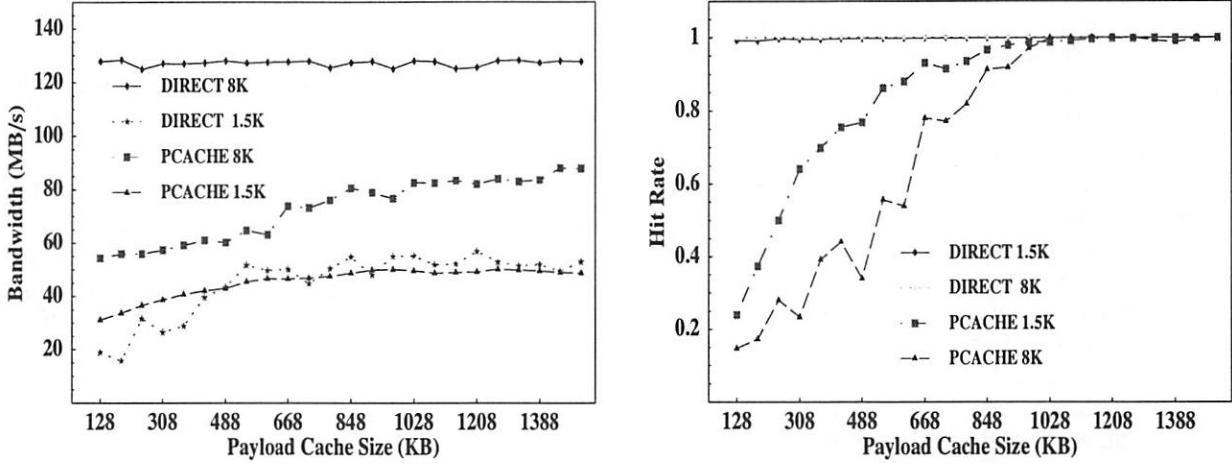


Figure 7: TCP bandwidth improvements with PCACHE and DIRECT for MTUs of 8KB and 1500 bytes, as a function of effective payload cache size up to 1.4 MB.

5.5 TCP Forwarding

We now show the effectiveness of payload caching for fast forwarding of TCP streams. For this experiment, we used *netperf* on four clients to initiate sixteen simultaneous TCP streams to a single server through a forwarding intermediary, with the interface Maximum Transmission Unit (MTU) configured to 1500 bytes or 8KB (Jumbo frames).

Figure 7 shows the resulting aggregate bandwidth and payload cache hit rate as a function of effective payload cache size. The graph does not show forwarding bandwidths without payload caching, since they are constant at the throughputs achieved with the smallest payload cache sizes. These base throughputs are 30 MB/s (240 Mb/s) with 1500-byte MTUs and 55 MB/s (440 Mb/s) with 8KB MTUs.

Using PCACHE, aggregate TCP bandwidth through the forwarder rises steadily as the payload cache size increases. With 1500-byte MTUs, payload caching improves bandwidth by 56% from 30 MB/s to a peak rate of 47 MB/s. With 8KB MTUs, payload caching improves bandwidth by 60% from 55 MB/s to 88 MB/s at the 1.4 MB payload cache size. It is interesting to note that these bandwidths are slightly higher than the peak bandwidths measured with UDP flows. This is because TCP's congestion control policy throttles the sender on packet drops. We measured slightly lower peak bandwidths for a single TCP stream; for example, a single stream with 8KB MTUs yields a peak bandwidth of 85 MB/s through a payload caching forwarder.

The right-hand graph in Figure 7 shows the payload cache hit rates for the same runs. Hit rates for the PCACHE runs rise steadily as the payload cache size increases, driving forwarding bandwidth up. For this experiment a megabyte of payload cache is sufficient to yield 100% hit ratios for all experiments.

Using direct forwarding (DIRECT) yields even higher peak bandwidths. A direct forwarder handles traffic at a full gigabit per second with 8KB MTUs, despite its I/O bus limitation. It might seem anomalous that bandwidth rises with larger cache sizes, even as the hit rate appears to be pegged at 100% even with small sizes. This effect occurs because all payload cache misses under DIRECT result in packet drops. Although a very small number of misses actually occur, they are sufficient to allow TCP's congestion control policy to quickly converge on the peak bandwidth achievable for a given cache size. With PCACHE, a payload cache miss only increases forwarding cost for an individual packet, which alone is not sufficient to cause TCP to throttle back until a queue overflows, forcing a packet drop. In all of our experiments, TCP congestion control policies automatically adjusted the send rate to induce peak performance from a payload caching forwarder.

6 Related Work

This section sets payload caching in context with complementary work sharing similar goals. Related techniques include peer-to-peer DMA, TCP splicing, and copy avoidance.

Like payload caching, peer-to-peer DMA is a tech-

nique that reduces data movement across the I/O bus for forwarding intermediaries. Data moves directly from the input device to the output device without indirection through the host memory. Peer-to-peer DMA has been used to construct scalable host-based IP routers in the Atomic project at USC/ISI [18], the Suez router at SUNYSB [16], and Spine at UW [11]. The Spine project also explores transferring the forwarded payload directly from the ingress NIC to the egress NIC across an internal Myrinet interconnect in a scalable router. Like DIRECT payload caching, this avoids both I/O bus crossings on each NIC's host, reducing CPU load as well. In contrast to peer-to-peer DMA, payload caching assumes that the ingress link and the egress link share device buffers, i.e., they are the same link or they reside on the same NIC. While payload caching and peer-to-peer DMA both forward each payload with a single bus crossing, payload caching allows the host to examine the data and possibly modify the headers. Peer-to-peer DMA assumes that the host does not examine the data; if this is the case then DIRECT payload caching can eliminate all bus crossings.

TCP splicing [8] is used in user-level forwarding intermediaries such as TCP gateways, proxies [17], and host-based redirecting switches [6]. A TCP splice efficiently bridges separate TCP connections held by the intermediary to the data producer and consumer. Typically, the splicing forwarder performs minimal processing beyond the IP layer, and simply modifies the source, destination, sequence numbers, and checksum fields in each TCP header before forwarding it. A similar technique has also been used in content switches [3], in which the port controller performs the sequence number translation. Once a TCP splice is performed, the data movement is similar to the NAT forwarding intermediary used in our experiments.

The primary goal of TCP splicing is to avoid copying forwarded data within the host. Similarly, many other techniques reduce copy overhead for network communication (e.g., Fbufs [7], I/O-Lite [15], and other approaches surveyed in [5]). These techniques are complementary to payload caching, which is designed to reduce overhead from unnecessary I/O bus transfers.

7 Conclusion

Data in the Internet is often forwarded through intermediaries as it travels from server to client. As network speeds advance, the trend in Web archi-

ture and other large-scale data delivery systems is towards increasing redirection through network intermediaries, including firewalls, protocol translators, caching proxies, redirecting switches, multicasting overlay networks, and servers backed by network storage.

Payload caching is a technique that reduces the overhead of data forwarding, while retaining the flexibility of host-based architectures for network intermediaries. By intelligently managing a cache of data payloads on the network adapter (NIC), the host can improve forwarding bandwidth and latency.

This paper shows how to incorporate payload caching into Unix-based frameworks for high-speed TCP/IP networking. It shows the interface between the host and the NIC and the new host functions to manage the payload cache. A key feature of our system is that the host controls all aspects of payload cache management and replacement, simplifying the NIC and allowing the host to use application knowledge to derive the best benefit from the cache. The NIC support for our payload caching architecture is straightforward, and we hope that future commercial NICs will support it.

Experimental results from the prototype show that payload caching and the direct forwarding extension improve forwarding bandwidth through host-based intermediaries by 40% to 60% under realistic scenarios, or up to 100% under ideal conditions. TCP congestion control automatically induces peak forwarding bandwidth from payload caching intermediaries. These bandwidth improvements were measured using effective payload cache sizes in the one-megabyte range on a gigabit-per-second network.

8 Availability

For more information please visit the website at <http://www.cs.duke.edu/ari/trapeze>.

9 Acknowledgments

Over the years many people have contributed to the development and success of the Trapeze project. Most notably Andrew Gallatin for his FreeBSD expertise and Bob Felderman at Myricom for his timely help. We thank the anonymous reviewers and our shepherd, Mohit Aron, for helpful critiques and suggestions.

References

- [1] Darrell Anderson, Jeffrey S. Chase, Syam Gadde, Andrew J. Gallatin, Kenneth G. Yocum, and Michael J. Feeley. Cheating the I/O bottleneck: Network storage with Trapeze/Myrinet. In *1998 Usenix Technical Conference*, June 1998.
- [2] Darrell C. Anderson, Jeffrey S. Chase, and Amin M. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, October 2000.
- [3] G. Apostolopoulos, D. Aubespain, V. Peris, P. Pradhan, and D. Saha. Design, implementation and performance of a content-based switch. In *Proceedings of IEEE Infocom 2000*, March 2000.
- [4] Jeffrey S. Chase, Darrell C. Anderson, Andrew J. Gallatin, Alvin R. Lebeck, and Kenneth G. Yocum. Network I/O with Trapeze. In *1999 Hot Interconnects Symposium*, August 1999.
- [5] Jeffrey S. Chase, Andrew J. Gallatin, and Kenneth G. Yocum. End system optimizations for high-speed TCP. *IEEE Communications, Special Issue on High-Speed TCP*, 39(4):68–74, April 2001.
- [6] A. Cohen, S. Rangarajan, and H. Slye. The performance of TCP splicing for URL-aware redirection. In *Proceedings of the 2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.
- [7] Peter Druschel and Larry L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 189–202, December 1993.
- [8] K. Fall and J. Pasquale. Exploiting in-kernel data paths to improve I/O throughput and CPU availability. In *USENIX Conference*, pages 327–334, January 1993.
- [9] Michael J. Feeley, William E. Morgan, Frederic H. Pighin, Anna R. Karlin, and Henry M. Levy. Implementing global memory management in a workstation cluster. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.
- [10] M. E. Fiuczynski, V. K. Lam, and B. N. Bershad. The design and implementation of an IPv6/IPv4 network address and protocol translator. In *Proceedings of USENIX*, 1998.
- [11] Marc E. Fiuczynski, Brian N. Bershad, Richard P. Martin, and David E. Culler. SPINE: An operating system for intelligent network adapters. Technical Report UW TR-98-08-01, Washington University, Department of Computer Science, September 1998.
- [12] Armando Fox, Steven D. Gribble, Yatin Chawathe, and Eric A. Brewer. Cluster-based scalable network services. In *Proceedings of Symposium on Operating Systems Principles (SOSP-16)*, October 1997.
- [13] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O’Toole Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the Fourth Symposium on Operating System Design and Implementation (OSDI)*, October 2000.
- [14] David F. Nagle, Gregory R. Ganger, Jeff Butler, Garth Gibson, and Chris Sabol. Network support for network-attached storage. In *Proceedings of Hot Interconnects*, August 1999.
- [15] V. S. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI’99)*, February 1999.
- [16] P. Pradhan and T. Chiueh. A cluster-based, scalable edge router architecture. In *Proceedings of the 1st Myrinet Users Group Conference*, 2000.
- [17] O. Spatscheck, J. Hansen, J. Hartman, and L. Peterson. Optimizing TCP forwarder performance. *IEEE/ACM Transactions on Networking*, 2(8):146–157, 2000.
- [18] S. Walton, A. Hutton, and J. Touch. Efficient high-speed data paths for IP forwarding using host based routers. In *Proceedings of the 9th IEEE Workshop on Local and Metropolitan Area Networks*, pages 46–52, November 1998.

A Waypoint Service Approach to Connect Heterogeneous Internet Address Spaces

T. S. Eugene Ng, Ion Stoica, Hui Zhang

Carnegie Mellon University

Pittsburgh, PA 15213

{eugeneng, istoica, hzhang}@cs.cmu.edu

Abstract

The rapid growth of the Internet has made IP addresses a scarce resource. To get around this problem, today and in the foreseeable future, networks will be deployed with reusable-IP addresses (a.k.a. private-IP addresses) or IPv6 addresses. The Internet is therefore evolving into a collection of networks of heterogeneous address spaces. Such development jeopardizes the fundamental bi-directional connectivity property of the Internet.

The problem is that, without IP addresses, non-IP hosts (i.e. reusable-IP or IPv6 hosts) cannot be directly addressed by IP hosts, making it impossible for IP hosts to initiate connections to them. To solve this problem, we propose a network layer waypoint (3rd-party network agent) service called AVES. The key idea is to *virtualize* non-IP hosts by a set of IP addresses assigned to waypoints. The waypoints then act as relays to connect IP hosts to non-IP hosts. This scheme allows every IP host to simultaneously connect to as many non-IP hosts as the number of waypoint IP addresses. Therefore high connectivity is achieved by AVES even when a small number of IP addresses are used. Unlike other known solutions, AVES can provide general connectivity and does not require any change to existing IP hosts or IP network routers for easy deployment. We have implemented and deployed an AVES prototype system at CMU. A wide range of applications have been shown to work seamlessly with AVES. Details of our implementation's design, performance and limitations are discussed.

1 Introduction

The Internet was originally conceived as a homogeneous global network in which all hosts would implement the network protocol Internet Protocol version 4 (IP or IPv4) [20],

This research was sponsored by DARPA under contract number F30602-99-1-0518, and by NSF under grant numbers Career Award NCR-9624979, ANI-9730105, ITR Award ANI-0085920, and ANI-9814929. Additional support was provided by Intel. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, NSF, Intel, or the U.S. government.

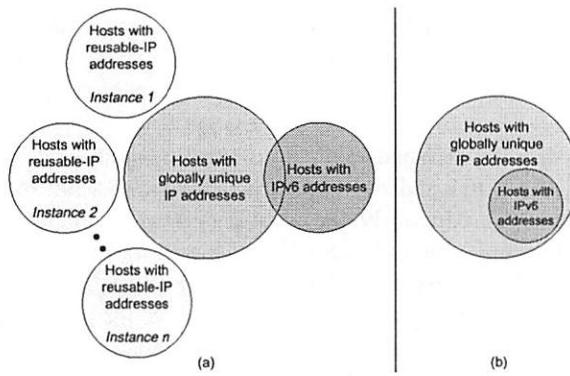


Figure 1: (a) Heterogeneous address spaces (b) IPv6 dual-stack strategy

have globally unique unicast IP addresses for identification and routing purposes, and could freely communicate with each other. But as the Internet evolves, it is becoming a heterogeneous network (as depicted in Figure 1(a)). In the process, *bi-directional* connectivity between hosts is lost. That is, given a pair of hosts, sometimes a connection can be established only if it is initiated by a particular side, and sometimes a connection cannot be established at all.

The root of the problem is that with the rapid growth of the Internet and the inefficient utilization of the IP address space, it has become clear that the relatively small 32-bit address space defined by IP is insufficient. The danger of exhausting the IP address space has prompted the Internet Assigned Numbers Authority (IANA) to conserve the remaining IP address space. This has resulted in two important development trends.

First, to get around the IP address shortage problem, it is increasingly common for networks ranging from large corporate networks to small home networks to be deployed using reusable-IP addresses.¹ By connecting a reusable-

¹Reusable-IP addresses are the network prefixes 10/8, 172.16/12 and 192.168/16 [21]. We use the term “reusable-IP addresses” instead of the more conventional “private-IP addresses” to distinguish from another use of these addresses to build *private* IP networks that are intentionally made inaccessible to the public Internet.

IP network to the IP Internet through a Network Address Translation (NAT) [25] gateway, *uni-directional* connectivity to the IP Internet is provided. That is, in general, reusable-IP hosts can initiate connections to IP hosts but not vice versa. Moreover, between two reusable-IP hosts belonging to different networks, there is generally no connectivity. Thus, hosts inside reusable-IP networks are not first-class Internet entities.

Second, as a long term solution, the IETF has designed the Internet Protocol version 6 (IPv6) [4] which defines an enormous 128-bit address space. Ideally, all new networks should now be deployed using IPv6, and all existing IP and reusable-IP networks should be upgraded to IPv6. However, since upgrading to IPv6 is a gradual process, IP and reusable-IP networks will remain in the foreseeable future. In addition, although new IPv6 networks can be fully compatible with IP when the dual-stack transition mechanism [8] is used, to achieve *full transparency*, every IPv6 host must be assigned an IP address and essentially behave as both an IPv6 and an IP host simultaneously as shown in Figure 1(b). Obviously, for many IPv6 network operators, this is simply not a viable option. Thus, a significant portion of IPv6 networks will likely be deployed as IPv6-*only* networks, and they will only have *uni-directional* connectivity to the IP Internet via Network Address Translation - Protocol Translation (NAT-PT) [27] gateways similar to the reusable-IP network scenario.

These development trends clearly indicate that the Internet today and in the foreseeable future will be a heterogeneous network composed of IP, IPv6 and reusable-IP address spaces as shown in Figure 1(a), and the fundamental bi-directional connectivity property of the Internet has been destroyed. In this environment, many common applications are no longer usable. Recent interest in peer-to-peer applications has raised awareness of this problem because under these applications there is no longer a distinction between client versus server and bi-directional connectivity is crucial. An important challenge is: *How can the lost connectivity in this heterogeneous environment be restored to as high a degree as possible?* The obvious difficulty is that, without IP addresses, non-IP hosts (i.e. reusable-IP or IPv6 hosts) cannot be directly addressed by IP hosts, therefore IP hosts cannot initiate connections to non-IP hosts directly. Any general solution to this problem must therefore allow a non-IP host to be identified by an identifier other than an IP address, and the identifier must be mapped to the actual non-IP host during communication.

To date, no known solution to this problem can provide *general* bi-directional connectivity and at the same time be deployed easily. Of the known solutions, which are discussed in Section 7, some are specific to one application (e.g. HTTP virtual hosting), some are application independent but require per application manual configurations and cannot provide general bi-directional connectivity (e.g. port forwarding), and some can provide general

bi-directional connectivity but require upgrades to existing IP hosts or IP network edge routers (e.g. SOCKS-based proposal). In practice, these upgrades to existing IP hosts or IP network edge routers are either too daunting to carry out, or there is no incentive to carry them out in the first place because they are aimed to benefit non-IP hosts and do not directly benefit existing IP hosts and networks.

In this paper, our aim is to design a solution that not only provides *general* bi-directional connectivity but also requires as little upgrades to existing software and hardware as possible. To achieve this goal, we propose a network layer waypoint service called AVES. Waypoints are 3rd-party network agents. The key idea is to *virtualize* non-IP hosts by a set of IP addresses assigned to waypoints. In this approach, we use DNS [15] names as identifiers for non-IP hosts and *dynamically* bind non-IP hosts to waypoint IP addresses during DNS name resolution in a *connection-initiator-specific* fashion. The waypoints then act as relays to connect IP hosts to non-IP hosts through AVES-aware NAT gateways.² This scheme allows every IP host to simultaneously connect to as many non-IP hosts as the number of waypoint IP addresses. As a result, high connectivity is achieved even when a small number of IP addresses are used. The internetworking heterogeneity is handled by the waypoints, no upgrade to existing IP hosts or IP network routers is required, making non-intrusive deployment of AVES possible. This approach is unique because it addresses an internetworking problem without changing the network layer of existing systems besides the NAT gateways.

It is important to note that AVES is optimized for deployment and is not perfect in every regard. In particular, AVES trades performance for deployability. It turns out that, since the binding of non-IP hosts to waypoint IP addresses during DNS name resolution is the critical step, the more control we have over the local DNS servers used by IP initiators, the better AVES can perform. However, in the extreme case where we have no control over the local DNS servers, AVES still provides the same connectivity but at the cost of lowered performance.

In Section 2, we further motivate the heterogeneous address space connectivity problem with a case study and precisely formulate the problem. We present the design of AVES in Section 3, and discuss its connectivity and deployability properties in Section 4. We have implemented a complete prototype of AVES on Linux and the details are presented in Section 5. In Section 6, we discuss key concerns about AVES such as application compatibility, scalability, and security. Finally, we discuss related work in Section 7 and summarize the paper in Section 8.

²Note that no known solution can provide general bi-directional connectivity without extending the functionality of the NAT gateway. However, since the operator of a NAT gateway has incentives to perform the upgrade, deployment should not be hindered.

		Responder		
		IP	IPv6	R-IP
Initiator	IP	Trivial	(a) Hard	(b) Hard
	IPv6	NAT-PT	Trivial	Reduces to (b)
	R-IP	NAT	Reduces to (a)	Reduces to (b)

Table 1: Taxonomy of address space connectivity

2 Case Study and Problem Formulation

To further motivate the need for bi-directional connectivity across heterogeneous address spaces, let us consider the DSL service at CMU. In April 1999, CMU began offering an internal DSL service that allowed users to obtain as many IP addresses as needed. Twenty months later, the 2000 IP addresses allocated to the service were exhausted. To conserve IP addresses, today only one statically assigned and one dynamically assigned IP address is provided per DSL line.

The situation has driven many of our DSL users to begin using NAT to get around the address allocation problem. Unfortunately with NAT, bi-directional connectivity is lost. This drastically affects the user's computing activities because fundamentally the university environment is not a pure client-server environment and bi-directional connectivity is critical. Although the DSL user will still be able to browse the web from home and access campus computing resources, she will not be able to remote login directly to her home computers using `ssh` or `telnet`. She also will not be able to host her own web servers or `ftp` servers on her home computers to distribute documents like digital videos and photos. When she is accessing campus computing resources from home, she also will not be able to bring up X Windows applications on her home computers (unless `ssh` X Windows connection forwarding is used). Many popular peer-to-peer applications also break down. For example, when both parties are behind NAT gateways, the popular music sharing software Napster will not work.

In Section 7, we discuss a simple port number forwarding work-around that can partially address these problems. However, this work-around works in the transport layer, requires per application manual configurations, and the connectivity achieved is unacceptable as only one home computer per port number can accept in-bound connections. In contrast, as we shall see, AVES is capable of allowing DSL users who deployed NAT gateways to fully regain all the above lost capabilities.

2.1 Heterogeneous Address Space Connectivity Problem

In the foreseeable future, three types of address spaces will coexist in the Internet, they are IP, IPv6, and reusable-IP. Table 1 describes the connectivity between all combinations of the three address space types. In a connection,

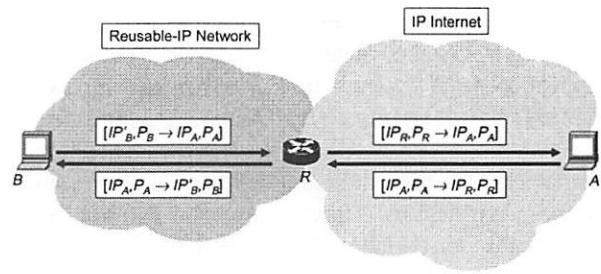


Figure 2: Out-bound connectivity via NAT gateway

the initiator is the “caller” host that sends the first packet to start the connection; the responder is the “callee” host that answers the in-coming connection. For example, to connect a reusable-IP (R-IP) initiator to an IP responder, it is well known that NAT [25] can be used and it works well in practice. Similarly, NAT-PT [27] can be used to connect an IPv6 initiator to an IP responder. On the other hand, to connect an IP initiator to an IPv6 responder (case (a), Table 1), or to connect an IP initiator to a reusable-IP responder (case (b), Table 1) is hard because the responder does not have any IP address and the initiator cannot address the responder directly. Solving these problems is the key challenge in maintaining the bi-directional connectivity abstraction of the Internet.

We emphasize that the problems underlying case (a) and case (b) are essentially identical, except that case (a) requires additional packet header format conversion which has been well documented in [18, 27]. Thus for simplicity, for the remainder of this paper, we only consider case (b), where an IP initiator is connecting to a reusable-IP responder. The results can be mapped to case (a).

Note that because there are multiple coexisting instances of the reusable-IP address space, connecting a reusable-IP initiator to a reusable-IP responder in a different instance of the address space is non-trivial. However, under NAT, this is equivalent to the initiator’s NAT gateway (which is an IP host) connecting to the reusable-IP responder. Therefore, this case can be reduced to case (b) as indicated in Table 1. Similarly, connecting a reusable-IP initiator to an IPv6 responder reduces to case (a), and connecting an IPv6 initiator to a reusable-IP responder reduces to case (b).

In summary, the key difficulty in achieving bi-directional connectivity across heterogeneous address spaces is to provide connectivity from IP hosts to non-IP hosts.

2.1.1 NAT and Its Limitation

It is helpful to understand the capability and limitation of NAT, but as we shall see, NAT can only provide unidirectional connectivity to the IP Internet. Figure 2 illustrates a typical scenario where a network is constructed using the reusable-IP address space and is attached to the IP Internet via a NAT gateway, R .

Assume R only owns a single IP address. Consider the

case where a reusable-IP host B (the initiator) is connecting to an IP host A (the responder). A reusable-IP address that belongs to host X is denoted IP'_X , and an IP address that belongs to host Y is denoted IP_Y . Assume B already knows the IP address of A .³ B simply initiates the connection by sending a packet to A . Suppose this is a TCP connection, and the packet sent by B has a source port number P_B and a destination port number P_A . We denote this packet by $[IP'_B, P_B \rightarrow IP_A, P_A]$ (the transport protocol is omitted for simplicity). The goal of NAT is to represent B in the IP Internet by R . As this packet is forwarded by R , R replaces IP'_B by its own IP address IP_R , and P_B by an available port number on R , say, P_R . The resulting packet is $[IP_R, P_R \rightarrow IP_A, P_A]$ and is forwarded out of the reusable-IP network. When a corresponding response packet $[IP_A, P_A \rightarrow IP_R, P_R]$ is received by R , R simply replaces the destination address by IP'_B and the destination port number by P_B . Since each 16-bit port number on R can be reused for different transport protocols, roughly 65,000 TCP and 65,000 UDP connections can be simultaneously active from initiating reusable-IP hosts to every port of every responding IP host even though R only has one IP address.

In contrast, if A is the initiator and B is the responder, the situation becomes very different. Because the only IP address owned by the reusable-IP network is IP_R , a DNS application level gateway [26] for in-bound NAT must resolve the name lookup for B to IP_R . Unfortunately, since IP_R can only refer to one reusable-IP host at any given time, with one IP address, NAT can only provide general in-bound connectivity to one responder in the entire reusable-IP network at a time. Since having one IP address is typical, NAT cannot provide acceptable in-bound connectivity.

3 AVES

In this section, we describe AVES (Address Virtualization Enabling Service), which can non-intrusively provide IP to IPv6 or IP to reusable-IP connectivity. Again, for simplicity, we only consider the reusable-IP scenario. The discussion also applies to the IPv6 scenario. For non-IP to IP connectivity, we simply rely on NAT and NAT-PT.

3.1 Overview

The key idea behind AVES is to *virtualize* non-IP hosts by a set of IP addresses assigned to waypoints. The waypoints then act as relays to connect IP hosts to non-IP hosts. Figure 3 illustrates this idea. In this example, there are two reusable-IP networks connected to the IP Internet, and the reusable-IP hosts B and C are virtualized by the IP addresses of waypoints W_2 and W_4 . As a result, IP initiators A and D can connect to responders B and C through the

³Such an IP address is usually obtained through DNS, and since a DNS server's address is known by configuration, we can assume any IP host's address can be known by B without any loss of generality.

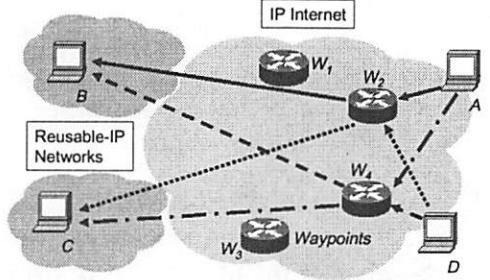


Figure 3: Overview of AVES

waypoints. Note that the bindings between non-IP hosts and waypoint IP addresses are *initiator-specific*. That is, each IP initiator has its own view. In our example, to A , B is bound to IP_{W_2} and C is bound to IP_{W_4} . On the other hand, to D , B is bound to IP_{W_4} and C is bound to IP_{W_2} . This ability to simultaneously bind an unlimited number of non-IP hosts to a waypoint IP address allows AVES to provide connectivity to an unlimited number of non-IP hosts. Another point worth noting is that the number of waypoint IP addresses only limits the number of non-IP hosts that each IP initiator can simultaneously connect to. Thus, for all practical proposes AVES requires only a small number of IP addresses, say a few tens, to achieve high connectivity.

More precisely, to implement AVES, a service provider deploys a small number of IP waypoints (W_1 - W_4) and AVES-aware DNS servers (not shown) for the reusable-IP domains. The waypoints have the following characteristics: (1) Waypoints are assigned IP addresses, possibly more than one per waypoint, in which case each IP address is logically a distinct waypoint. Here we assume only one IP address is assigned per waypoint. (2) Waypoints are capable of performing address (and protocol, in the case of IPv6) translation, they serve as relays for traffic crossing heterogeneous address spaces. (3) Because waypoints are network agents, they can be deployed non-intrusively without global coordination. Under AVES, for IP initiator A to connect to reusable-IP responder B , it first performs a DNS name lookup for B ; this marks the beginning of a *session*. The name lookup operation serves two purposes. First, the DNS name will uniquely identify the responder even though it does not have a unique IP address. Second, when the DNS query is processed by an AVES-aware DNS server, the non-IP host is bound to the IP address of a chosen waypoint, in this case IP_{W_2} . Again, this binding is initiator-specific so that a waypoint IP address can be bound to multiple non-IP hosts simultaneously. Instructions are then sent by the AVES-aware DNS server to W_2 so that it can correctly relay packets. IP_{W_2} is returned to A in the DNS reply with the time-to-live field set to zero (i.e. no caching of IP address records is allowed; however name server records can be cached). The session is now established, and A can open arbitrary connections to B through W_2 . A session is terminated when a timeout occurs after

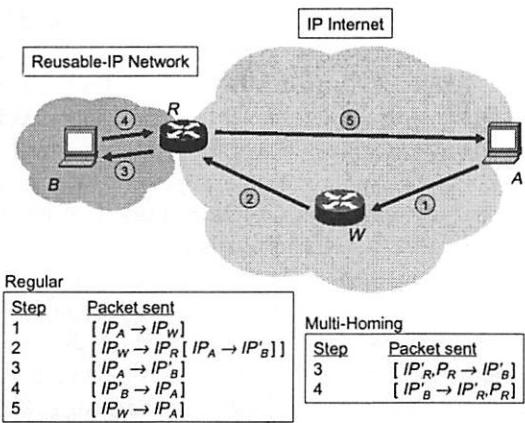


Figure 4: Data path operations

a period of inactivity. Afterwards, an initiator can regain connectivity by starting another session. This example illustrates several key ideas underlying AVES:

- **Virtual expansion of IP address space** – A waypoint IP address can virtually represent, or virtualize, an unlimited number of non-IP responders in the IP Internet simultaneously because the binding is *initiator-specific*. Hence, AVES virtually expands the IP address space, achieving high connectivity when only a small number of IP addresses are used.
- **Heterogeneity hiding** – From the point of view of an IP initiator, with AVES, all responders appear to be IP hosts with IP addresses. Thus, there is no need to modify existing IP hosts or IP network routers to achieve connectivity.
- **Transparent access** – An initiating IP host accesses AVES transparently via DNS host name resolution. The IP address of the selected waypoint is returned to the initiating IP host. The service abstraction provided by AVES is therefore simply an IP address, which is most compliant with existing applications.

In the following, we first explain the data path operations, then we explain the control path operations for configuring the data path and discuss deployment scenarios in relation to our case study in Section 2. The connectivity achieved by AVES is summarized precisely in Section 4.

3.2 Data Path Operations

Figure 4 shows a typical data path between an initiator A and a reusable-IP responder B . W is a waypoint and R is an AVES-aware NAT gateway. W virtualizes B for A . Thus, to A , the IP address of B is IP_W . To correctly relay packets from A to B , W has been configured by an AVES-aware DNS server via the control path protocol described in Section 3.3 with the following translation table entry (we omit the port numbers as they are unimportant):

Original packet	Translated packet	Encapsulation header
$[IP_A \rightarrow IP_W]$	$[IP_A \rightarrow IP'_B]$	$[IP_W \rightarrow IP_R]$

That is, when a packet from IP_A is received by W (recall that the binding is *initiator-specific*), the destination address of the packet is translated to IP'_B , and the resulting packet is tunneled from IP_W to IP_R . Note that we propose a tunneling based mechanism here despite the header overhead because the encapsulation header allows complete information about the session to be carried along with each data packet so that R can process each in-coming data packet purely based on its packet headers. This eliminates the need for a control path mechanism to configure R ahead of time, resulting in a simpler protocol. In the following, we describe two versions of the data path operations. The first version applies when the reusable-IP network is connected to the IP Internet via a single NAT gateway. The second one applies when the reusable-IP network is “multi-homed”, that is, it is connected to the IP Internet via multiple NAT gateways.

The data path operations without multi-homing support are as follows. A initiates a connection to B by sending the packet $[IP_A \rightarrow IP_W]$ (step 1). When W receives such a packet, it transforms the packet into $[IP_A \rightarrow IP'_B]$ and encapsulates the packet with the header $[IP_W \rightarrow IP_R]$. We denote the final packet by $[IP_W \rightarrow IP_R[IP_A \rightarrow IP'_B]]$. To enhance security, this packet is authenticated by W . The packet is then forwarded (step 2) and later received by R . In addition to supporting the basic functionalities of a NAT gateway, R is extended such that when R receives an authentic encapsulated packet from W , it first determines whether a packet of the same connection (matching addresses in both outer and inner packet headers and port numbers, if any) has been seen before. If not, R creates a local translation table entry such that, when a corresponding out-bound packet $[IP'_B \rightarrow IP_A]$ (with matching port numbers, if any) is received, it will modify this out-bound packet to $[IP_W \rightarrow IP_A]$ before forwarding it out of the reusable-IP network. After creating this translation table entry, R removes the encapsulating packet header from the in-coming packet and forwards the inner packet to B (step 3). Finally, when B sends a reply to A (step 4), the packet $[IP'_B \rightarrow IP_A]$ is modified by R to $[IP_W \rightarrow IP_A]$ and then forwarded to A (step 5). Through these mechanisms, a connection from A to B is established.

The operations above prevent a reusable-IP network from being multi-homed because they do not guarantee that the out-bound packets of a session will traverse the same NAT gateway as the in-bound packets, consequently out-bound packets might not be translated correctly. To accommodate a multi-homed network, we modify the data path operations as follows. In step 3, the source address of an in-bound packet is translated to the reusable-IP address of R (IP'_R), and the source port number is translated to a chosen number (P_R) to maintain the binding. The resulting packet for step 3 is $[IP'_R, P_R \rightarrow IP'_B]$, and the packet for step 4 is $[IP'_B \rightarrow IP'_R, P_R]$. As a result, out-bound packets are guaran-

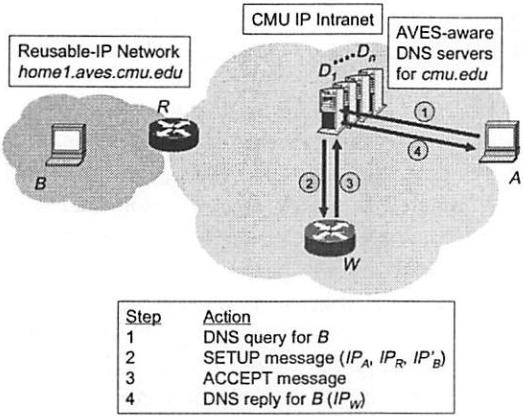


Figure 5: Control path operations for intranet deployment

ted to traverse the same border gateway as in-bound packets. For an in-bound ICMP [19] query packet, the Identifier field can be translated instead since there is no port number.

We have implemented both variations of the data path in our prototype system, see Section 5 for details. Limitations of these mechanisms are discussed in Section 6.

3.3 Control Path Operations

The AVES control path mechanisms are used to configure the data path. That is, when a DNS query for a reusable-IP responder is processed by an AVES-aware DNS server, a waypoint is selected to virtualize the reusable-IP responder and the appropriate translation table entry is installed at the selected waypoint so it can correctly relay packets.

It is important to recall that the bindings between waypoint IP addresses and reusable-IP responders must be *initiator-specific*. This allows a waypoint IP address to simultaneously virtualize many reusable-IP responders, and as a result high connectivity can be achieved with a small number of IP addresses assigned to waypoints. Unfortunately, creating initiator-specific bindings is not easy because the IP address of the initiator is typically not available in a DNS query received by an AVES-aware DNS server. This is because, in practice, virtually all end host systems implement *recursive* DNS query [15]. That is, an end host sends a *recursive* DNS query to its local DNS server, and this local DNS server generates additional *iterative* queries on behalf of the end host, and eventually returns the answer to the end host. Thus, an AVES-aware DNS server generally only interacts with the local DNS server of the initiator, the IP address of the initiator is obscured. In the following, we describe two deployment scenarios and the appropriate techniques in each case to create initiator-specific bindings.

3.3.1 Scenario 1 – Intranet Deployment

Let us reconsider the scenario discussed in Section 2. CMU can deploy AVES to restore bi-directional connectivity

within the CMU intranet so that DSL users will be able to access their home computers directly from any host within the CMU intranet. To do so, CMU would deploy waypoints and upgrade its local DNS servers to make them AVES-aware. By upgrading the local DNS servers, initiator-specific bindings can be created easily since an initiator's IP address is now available in the IP headers of its DNS queries to the AVES-aware local DNS servers.

Figure 5 shows how this scheme works. Under this scheme, reusable-IP networks will use a common domain name suffix, say `aves.cmu.edu`, for easy identification. In our example, the reusable-IP network has a domain name `home1.aves.cmu.edu`. $D_1 - D_n$ are upgraded AVES-aware local DNS servers. The control path operations are as follows. Initiator A 's DNS query for B is directly sent to one of the AVES-aware local DNS servers, D_1 (step 1). D_1 is by configuration aware of the IP address of the AVES-aware NAT gateway R and the reusable-IP address of B . Upon receiving the DNS query, D_1 selects at random a waypoint among a set it knows, in this case W , and sends a SETUP message to W (step 2).⁴ The SETUP message contains IP_A , IP_R , and IP'_B , which are necessary to create a data path translation table entry on W . When W receives the SETUP message, it examines its data path translation table to see if it can accept the request. Let us denote a translation table entry E on W more compactly by $\{IP_{initiator}, IP_{NAT}, IP'_{responder}\}$. Then, W can accept the request for initiator IP_A , NAT gateway IP_R , and responder IP'_B if and only if,

$$\forall E, IP_{initiator} = IP_A \Rightarrow (IP_{NAT}, IP'_{responder}) = (IP_R, IP'_B).$$

That is, if W already has a translation table entry for initiator IP_A , and the responder of that entry is not the same as the one in the SETUP message, then W must reject the request and reply with a REJECT message because W cannot be used to relay a particular initiator to more than one responder. On receiving a REJECT message, for simplicity, the AVES-aware DNS server will simply do nothing and let the initiator perform the DNS name lookup again to retry. In our example, the admission control criterion is satisfied, so W accepts the request, creates the corresponding translation table entry, and sends back an ACCEPT message (step 3). Finally, when D_1 receives the ACCEPT message, it responds to A 's DNS query for B with the IP address of the selected waypoint, IP_W , with the time-to-live field set to zero (step 4). Note that the messages between waypoints and the AVES-aware DNS servers are authenticated to prevent unknown sources from gaining control of the system. Also, the messages can be lost in the network. Waypoint failure and packet loss are simply handled by initiator A 's DNS query timeout/retry mechanism. Limitations of this scheme are discussed in Section 6.

⁴Selecting a waypoint based on performance metrics is a topic for future research.

```

 $X \leftarrow \{\}$ ;
on receiving new connection packet [ $IP_S \rightarrow IP_W$ ] :
  if  $IP_S$  should be rejected
    discard packet;
    return;
  if  $\exists E$  s.t.  $IP_{initiator} = IP_S$  and
     $(IP_{NAT}, IP_{responder}) \neq (IP_R, IP_B)$ 
    /* violation */
    discard packet;
     $X \leftarrow X \cup \{IP_S\}$ ;
  else
    accept packet;
    if  $\nexists E$  s.t.  $IP_{initiator} = IP_S$ 
      create  $E = \{IP_S, IP_R, IP_B\}$ ;
on exiting wait state after  $T_{wait}$  :
  reject connections from  $\forall IP_S \in X$  for  $T_{reject}$ ;

```

Figure 6: Waypoint wait state algorithm for general deployment

3.3.2 Scenario 2 – General Deployment

There are two major disadvantages of the previous deployment scheme. First, reusable-IP hosts are still unreachable from hosts that do not belong to CMU’s intranet. Second, deployment requires upgrading CMU’s local DNS servers and thus requires CMU’s consent.

It is possible to overcome both of these short-comings by using a technique called *delayed binding* at the expense of lowered performance. The basic idea is that, a waypoint does not need to know the identity of the initiator to accept a request. It can accept the request optimistically and wait for the connection from the initiator to arrive, and only at that time admission control is performed and the actual binding is created.

Under this scheme, reusable-IP networks will use a common domain name suffix that is independent of any organization, say `avesnet.net`. Waypoints and AVES-aware DNS servers are independently deployed for the `avesnet.net` domain. No upgrade to any existing DNS server is needed. When a DNS query is received by an AVES-aware DNS server for `avesnet.net`, although the initiator’s IP address (IP_A) is no longer known, the AVES-aware DNS server can still select a waypoint W and send it a SETUP message containing IP_R , IP_B' , and IP_{DNS} (the IP address of the initiator’s local DNS server). Without knowing IP_A , W can no longer perform the admission control test stated in Section 3.3.1. However, W can make use of whatever information it has and decide whether to accept the request (in the simplest case, W always accepts the request). If W accepts the request, it replies with an ACCEPT message, and immediately enters a *wait state* for a short period of time, T_{wait} , and executes the algorithm shown in Figure 6. During this time, W does not accept other in-coming SETUP requests. Thus, requests are serialized.

In summary, during this wait state, when a new connection from some initiator S arrives (indicated by a TCP SYN packet or any non-TCP packet), S is potentially the initia-

tor that W is waiting for. Thus, W checks to see if S violates the admission control criterion (note that E in Figure 6 denotes a waypoint translation table entry as defined in Section 3.3.1). If so, the packet must be rejected, and S is recorded in the set X of violators. If later a new connection from initiator A arrives, and A does not violate the admission control criterion, and W has no existing translation table entry for A , then a new translation table entry is created for A and bound to responder B . Upon exiting the wait state, connections from initiators in X must be rejected for a time period T_{reject} to force these initiators to retry their connections. Note that T_{reject} should not be too large or it may negatively affect future requests from the same initiator.

We have fully implemented delayed binding in our prototype system and it works well (see Section 5 for details). Since this technique is independent of organizational boundaries, it is actually feasible for our prototype system to provide service to reusable-IP networks outside of CMU.

One disadvantage of delayed binding is that connections need to be retried whenever an admission control violation is committed. Fortunately, when the number of waypoints is greater than the average number of simultaneous sessions opened by an initiator, the chance of this can be kept small. Another disadvantage is that the peak rate at which the whole system can accept new sessions is limited to N/T_{wait} sessions per second, where N is the number of IP addresses assigned to waypoints. Our prototype system, with 50 IP addresses and a T_{wait} of 2 seconds, can accept 25 sessions per second. While this is quite reasonable for CMU’s DSL users, we do not advocate the use of our system to serve a popular web server. Other limitations regarding security and state consistency are discussed in Section 6.

3.3.3 Final Comment

Note that if we can extend the DNS protocol to always carry the original initiator’s IP address in all DNS queries, deployment of AVES can be greatly simplified. General deployment can be achieved without making existing DNS servers AVES-aware or using delayed binding.

4 AVES Connectivity and Deployability

In Section 3.2, we described two data path designs. Assuming N IP addresses are assigned to waypoints, the in-bound connectivity achieved by each design is as follows. For the regular data path design without support for multi-homing:

- All hosts in non-IP networks are simultaneously reachable directly by IP hosts, regardless of the size of N .
- Each IP host can simultaneously open N sessions to reach a maximum of N non-IP hosts.

With multi-homing support, since port or identifier numbers are used for connection demultiplexing, the following additional restriction is imposed:

- Each port number of each non-IP host can be reached by no more than 65,000 TCP and 65,000 UDP connections simultaneously through each AVES-aware NAT gateway. Also, through each AVES-aware NAT gateway, each non-IP host can be reached by no more than 65,000 ICMP connections simultaneously. If a protocol does not use port or identifier number, then each non-IP host can only be reached by one connection of such protocol through each AVES-aware NAT gateway at a time.

Thus, as long as N is greater than the average number of simultaneous *sessions* to non-IP hosts opened by a typical IP initiator, say $N = 50$, in-bound connectivity can be restored to a high level.

To summarize AVES's deployability, waypoints can be independently deployed; NAT gateways need to be extended, however this is necessary and acceptable because their operators have the right incentives to perform the upgrade. To deploy AVES for an intranet, upgrading the local DNS server software will provide the best performance. However, even when it is impossible to upgrade existing DNS servers, the delayed binding technique can be used at the expense of lowered performance. In all cases, no existing IP hosts or IP network routers need to be modified.

5 Implementation

For fast prototyping and simple deployment, we have implemented AVES for reusable-IP networks as a suite of user-level software on the Linux platform. The three components are (1) the AVES-aware DNS server daemon, (2) the AVES waypoint daemon, and (3) the AVES NAT gateway daemon. To enhance security, data and control messages between the three components are authenticated by including with a message the 16-byte MD5 checksum [22] of the message together with a 16-byte secret key. One secret key is shared between the AVES-aware DNS servers and waypoints while each AVES-aware NAT gateway has a specific secret key. We save the discussion of some safeguarding security features until Section 6.2. In the following, we describe the three individual components, then we report some performance figures. Finally, we describe our current prototype system.

5.1 AVES-Aware DNS Server Daemon

Our AVES-aware DNS server daemon is based on the named DNS server in the BIND 8.2.3 distribution [11] and runs on a Linux PC. We modified named to intercept any outgoing DNS reply message containing a DNS name with the `avesnet.net` suffix because such a reply contains the

reusable-IP address of the named responder. This is accomplished by inserting a function call in `ns_req()` (to intercept answers from the local cache) and `ns_resp()` (to intercept answers from other origins). Once a reply is intercepted, a lookup table is consulted to obtain the IP address of the reusable-IP domain's NAT gateway and a waypoint IP address is chosen. NAT gateway IP addresses are obtained from the NAT gateways periodically to accommodate dynamic address assignment (see Section 5.3 for more details), while the waypoint IP addresses and the reusable-IP host addresses are kept in configuration files. A SETUP message with a unique serial number is then sent via UDP to the chosen waypoint, the intercepted DNS reply is altered to contain the chosen waypoint IP address and is set aside. When the corresponding ACCEPT message is received from the waypoint, the DNS reply is finally sent to the requester. DNS replies that have been set aside are removed if the corresponding ACCEPT messages are not received within 3 seconds.

5.2 AVES Waypoint Daemon

Our AVES waypoints are based on Linux PCs. Each machine can be assigned multiple waypoint IP addresses as aliases of its network interface. The AVES waypoint daemon uses the Linux IP firewall (`ipfw`) API to filter selected data packets to user-level for manipulation, it requires Linux kernel version 2.2 or higher. To filter incoming data packets to user-level, the waypoint daemon opens a raw NETLINK_FIREWALL netlink socket. Filter entries can then be added to the input firewall via the `ipfw` API and the kernel can be instructed to direct matching packets to the netlink socket. After data packets are manipulated in user-level, they are reinjected into the network via a raw socket with the IP header included option (IP_HDRINCL) enabled.

We have fully implemented the delayed binding technique as described in Section 3.3.2. When there are multiple alias waypoint IP addresses on the machine, each address is treated independently by the waypoint daemon. The wait period T_{wait} in our implementation is 2 seconds which should provide sufficient time for a connection to be made. When the waypoint IP address IP_W is in a wait state, the waypoint daemon filters all in-coming packets with destination address IP_W regardless of the source address. Packets that do not indicate a new connection are processed normally according to existing translation table entries. A new connection (indicated by a TCP SYN packet, or any non-TCP packet) to IP_W is either accepted or rejected according to the algorithm shown in Figure 6. If the connection is accepted, a filter for the source and destination address pair is added to the firewall and a translation table entry is created. The packet is then processed normally. If the connection is rejected, the packet is dropped, and an ICMP "destination host unreachable" message [19] is sent back to the initiator. This signals to the initiator that

it needs to retry the connection. The reject period T_{reject} is 3 minutes in our implementation, which we think is sufficient to prompt the initiator to retry the connection, and does not make IP_W unavailable to the initiator again for too long. Note that, when IP_W is in a wait state, AVES SETUP messages sent to IP_W are ignored for simplicity. Below is a summary of the other noteworthy features supported by the waypoint daemon:

Fragmentation & Path MTU Discovery – Because the waypoint daemon encapsulates a translated packet in an IP header and adds a 16-byte MD5 checksum, typical 1500 byte in-coming Ethernet packets will have to be fragmented on their way out. It turns out that Linux does not perform fragmentation for packets sent through a raw socket with the IP_HDRINCL option enabled, therefore IP fragmentation has been implemented in the waypoint daemon. The waypoint daemon also supports path MTU discovery [16]. That is, when the “Don’t Fragment” flag of an in-coming IP packet is set but fragmentation is necessary, the waypoint daemon drops the packet, and returns an ICMP “destination unreachable fragmentation needed” message [19] to the initiator with the MTU field set to 1464 bytes. Finally, a consequence of IP fragmentation is that, the AVES NAT gateway must be configured to reassemble all in-coming fragmented packets so that the AVES NAT daemon can function properly.

Protocol Specific Timeouts – A translation table entry represents a session opened by an initiator and will expire if there is no traffic activity for a period of time. To optimize resource usage, we use different timeout values for different protocols. The protocols the waypoint daemon recognizes are ICMP, TCP, and UDP. First, if an initiator is transmitting an unknown protocol or a mixed set of protocols to the responder, a default timeout value of 15 minutes is used. For ICMP, since it is mostly generated by ping or traceroute, we aggressively timeout these entries in 1 minute. For UDP, the timeout value is set to 15 minutes. For TCP, the timeout value is set to 30 minutes. These choices are somewhat arbitrary, but we think they are reasonable. To further optimize, we keep track of the TCP connections that correspond to a translation table entry, and when all of them have terminated (indicated by TCP FIN packets), the translation table entry is removed immediately without waiting for the timeout. An exception to this is when the traffic is HTTP (i.e. port 80) because popular browser software such as Netscape and Internet Explorer always cache DNS replies for 15 minutes. Thus, for HTTP, we simply use a timeout value of 20 minutes without checking for TCP FIN packets.

5.3 AVES NAT Daemon

Our AVES-aware NAT gateways are based on Linux PCs as well, and they are assumed to be already configured to perform defragmentation of in-bound packets and IP masquerading (i.e. out-bound NAT), which is fully compatible

with AVES. Similar to the waypoint daemon, the AVES NAT daemon also filters selected packets to user-level for manipulation. To handle NAT gateway dynamic IP address assignment, periodically, the NAT daemon sends authenticated registration messages via UDP to the AVES-aware DNS servers to report its current IP address. These messages are sent more frequently when an address change is detected to ensure with high probability that the update is completed promptly.

The basic operations performed by the NAT daemon is as described in Section 3.2. The NAT daemon by default filters all in-coming encapsulated packets. When an authentic encapsulated packet is received and the connection has not been seen before, a filter is installed for the corresponding out-bound packets, and a translation table entry is created. Several other noteworthy features of the NAT daemon are summarized below:

Protocol Specific Timeouts – Similar to the waypoint daemon, different timeout values for the translation table entries are used for different protocols. The policy is exactly the same as that in the waypoint daemon.

ICMP Handling – For traceroute, even though the in-bound packet is UDP, an out-bound ICMP packet is triggered. To support traceroute, when an in-bound UDP connection is received, we install an extra filter and translation table entry for the potential out-bound ICMP packets. The timeout is set to 5 seconds so that if no ICMP packets are triggered, the state is removed quickly. In addition, since many ICMP message types carry IP addresses and port numbers in the packet payload, the AVES NAT daemon translates the payload accordingly as well.

Multi-Homing Support – To support multi-homing as described in Section 3.2, the source address and port number (or ICMP Identifier) of an in-bound packet are translated. To choose a suitable port number, we simply pick a port number between 1024 and 65535 at random, and test to see if that port number can be bound to a TCP and a UDP socket. The process repeats until a port number that is free is found. This makes sure that our port number allocation will not interfere with the other operations of the NAT gateway. However, notice that our straight-forward implementation does not achieve the theoretical highest connectivity as discussed in Section 4. In our implementation, only 64,512 in-bound connections (TCP or UDP) can be simultaneously active regardless of the destinations of the connections. This is however more than sufficient for the purpose of our prototype.

Limitations of Multi-Homing Support – When multi-homing is enabled, only one reusable-IP network can be connected to a NAT gateway because when there are multiple reusable-IP networks attached, our implementation is not yet capable of translating the source address of an in-bound packet to the address of the correct output network interface. Also, some applications, most notably ftp, will not work when multi-homing is enabled because the

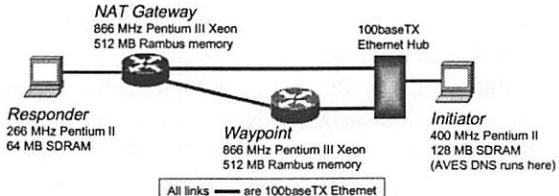


Figure 7: Performance measurement testbed

reusable-IP ftp server cannot open the data connection to the IP client since the IP client's address has been translated. And because the server passes its reusable-IP address, say 10.0.0.1, to the client, even if "passive mode" [1] is enabled, the client will attempt to open the data connection to the address 10.0.0.1 instead of the waypoint IP address. The only way to get around this is to translate ftp control packets' content.

5.4 Performance

To measure the performance of our system, we set up a small 100 Mbps Ethernet testbed as shown in Figure 7. For data path performance, we instrumented the Linux kernel version 2.2.14 and our daemon software to measure, with the Pentium CPU cycle counter, the processing time of a packet in the waypoint and the NAT gateway (both in-bound and out-bound directions). We measured three quantities: (1) the total packet processing time from the moment `netif_rx()` was called by the Ethernet device driver after receiving a packet until the moment `dev_queue_xmit()` was called to pass a processed packet to the device driver for transmission; (2) the AVES daemon processing time from the moment a packet was received by a daemon socket until the moment before the processed packet was sent out on a socket; (3) the time spent on computing the MD5 authentication checksum in the AVES daemon. We sent UDP packets of varying sizes between the initiator and the responder and recorded the processing times. Our experiments show that all the processing times scale linearly as the packet size varies. Figure 8 shows the partial results, averaged over 10,000 packets, for the smallest (36 bytes) and largest (1464 bytes) packet sizes we have tried.

There are several noteworthy points. First, implementing our software in user-level adds a very significant overhead due to the memory copies and context switches. We can expect a kernel-level implementation of our software will have a total processing time very close to the AVES daemon processing time. Second, almost all the AVES daemon processing time is spent on computing the MD5 authentication checksum (note that no authentication is needed for out-bound packets at the NAT gateway). This overhead can be reduced if we only authenticate the packet headers but at the cost of lowered security. Finally, based on these measurements, our software can theoretically sustain a throughput of 233 Mbps with 1464 byte packets in our testbed.

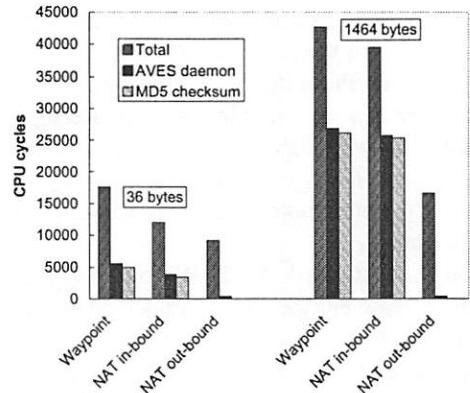


Figure 8: Packet processing times

We have also conducted end-to-end throughput experiments. When we sent 1464 byte packets from the initiator to the responder, the throughput was limited only by the link capacity, as the system achieved 96 Mbps with UDP and 80 Mbps with TCP. However, when we sent 48 byte packets, our software was only able to achieve 41 Mbps with TCP. This is actually higher than the calculated maximum of 19 Mbps based on the processing times measurements due to the amortization of kernel overheads over a sequence of packets. We expect the throughput with UDP to be slightly better; however, due to a device driver bug with the Intel EtherExpress Pro 100 network interface card, we were unable to send 48 byte UDP packets faster than 10 Mbps without causing the interface card to shutdown.

Next we measured the performance of the control path. Typically, the time required to resolve an AVES DNS name is dominated by the network delays of the DNS and AVES control messages. To factor out the network delays, we ran both the AVES-aware DNS server daemon and the waypoint daemon on the initiator machine. A program running on the initiator that repeatedly issued `gethostbyname` system calls for the responder was used to drive the system. We then measured the number of CPU cycles, including socket reads and writes, averaged over 20 requests, consumed by each control path component.

We found that the total time required to complete a `gethostbyname` system call was on average 357,000 cycles. This total time can be further broken down as follows. First, it took 142,000 cycles to process the DNS query at the AVES-aware DNS server daemon and send the SETUP message to the waypoint daemon. Second, the waypoint daemon took 71,000 cycles to process the SETUP message and send back the ACCEPT message. Finally, another 17,900 cycles were spent at the AVES-aware DNS name server daemon to process the ACCEPT message and send out the final DNS reply message. Computing the MD5 authentication checksum of an AVES control message took 3,100 cycles. Thus, the DNS query processing is the bottleneck. With a 400 MHz AVES-aware DNS server, at most 2,800 sessions can be set up per second. Of course

if delayed binding is used, the protocol will impose a much stricter limit.

5.5 Prototype System

We have registered the domain name suffix `avesnet.net` and deployed an AVES prototype system. A Linux PC serves as the AVES-aware DNS server for the `avesnet.net` domain. Two other Linux PCs serve as waypoints, each with 25 IP aliases for a total of 50 waypoint IP addresses. We currently have ten trial subscribers. Subscribing to AVES is a simple three step process. A reusable-IP network operator needs to (1) obtain a sub-domain under `avesnet.net` from the service operator, (2) inform the service operator the desired DNS name to reusable-IP address mappings, and (3) run the AVES NAT gateway daemon.

Using our prototype, we have shown that a diverse set of applications work seamlessly with AVES. We are able to remote login from any IP host to a demo reusable-IP host called `demo1` using `telnet` or `ssh`, perform file transfers using `ftp` (when multi-homing is disabled, using non-passive mode) or `scp`, export a NFS file system on `demo1` and mount the file system on any IP host. We are also able to host a web server on `demo1` and access the content from any IP host. When logging in from `demo1` to an IP server (by out-bound NAT), we are able to directly bring up X Windows applications on `demo1` after the `DISPLAY` environment variable has been correctly set. Diagnostic tools such as `ping` and `traceroute` also work transparently (with limitations described in Section 6). An on-line demo of our prototype can be found at <http://www.cs.cmu.edu/~eugeneng/research/aves/>.

6 Discussion

In this section, we discuss the limitations imposed by our approach. This is by no means an exhaustive account. It is important to realize that AVES is making a trade-off between non-intrusively restoring bi-directional connectivity to a high degree and the limitations it imposes. We believe that most of the limitations can be coped with, and the benefits of AVES significantly out-weigh the limitations.

6.1 Application Requirements

There are three types of limitations imposed by AVES that may conflict with an application's behavior, they are (1) limitations due to address translation, (2) limitations due to the need for session creation, and (3) limitations due to the need for consistent state maintenance. In the following, we discuss what rules must an application obey in order to be compatible with AVES.

The first type of limitation is not specific to AVES, but it is a fundamental limitation of any address translation scheme such as NAT, TRIAD [3] and IPNL [6] (discussed in Section 7). The main problem is that some applications break the layering semantics by exchanging lower layer information such as IP addresses and use the information directly. In [10] and [24], some NAT-friendly application design guidelines are given. Because AVES also performs address translation, some of these guidelines are relevant (guidelines that aim to avoid in-bound connections are no longer needed under AVES). Specifically, with respect to address translation, in order to be compatible with AVES coupled with NAT, an application should not pass IP addresses in the packet payload; instead, DNS names should be passed, and name resolution should always be used to determine the IP addresses. Listener port number passing is actually no longer a problem if DNS names are used. Also, applications should not expect the network and transport headers to be unmodified in transit. Clearly IPsec [13] would not work across NAT or AVES. In IETF, there is ongoing work on making NAT more IPsec-friendly [2]. In Section 6.4, we will also describe a change to the AVES data path that may make AVES more IPsec-friendly.

AVES fundamentally requires a session to be opened by an initiator before connectivity is provided. Therefore, an application must perform a DNS lookup before communication begins. Moreover, communication must begin immediately after the DNS lookup to work with delayed binding since T_{wait} is typically small. When a connection is rejected, the application must perform a DNS lookup again to restart the session. Note that a T_{wait} of 2 seconds used by our prototype might not work for an application like `traceroute`, since it progressively probe the network hop-by-hop and this process may take more than 2 seconds to reach the waypoint.

Finally, an application must obey some rules to maintain consistency between its state and the waypoints' state. From a waypoint's point of view, a session is terminated when an idle timeout occurs, or when all connections of the session (assuming they are all TCP) are terminated. Therefore, an application must send periodic keep-alive messages. In addition, it must not reuse DNS lookup results across sessions (as in the web browser example). An application must also begin communication within T_{wait} after a DNS reply is received, otherwise, the application's view is stale. When a connection is rejected, an application must also restart the session by performing another DNS lookup. These rules will prevent the initiator from having a stale view. If an application does not follow these rules, then it may have a stale view, in that case, there are two possible outcomes. First, the connection may get rejected by the waypoint because it has no state for the initiator. Second, the connection may get relayed to the wrong responder because the waypoint has other state for the same initiator. On the other hand, a waypoint may have a stale view if

a session has ended (e.g. a UDP session is terminated by the application) but it still keeps state about it. This type of inconsistency only affects performance, not correctness, because it simply makes the waypoint unavailable to the same initiator for a longer period.

6.2 Security

An obvious concern with AVES is whether it is secure. Can attackers flood the system? Will AVES reusable-IP hosts be exposed to attackers at the level of regular IP hosts? Can attackers cause the system to mis-behave? In the following, we discuss these issues in detail. We assume a general deployment scenario where delayed binding is used since a secure environment is assumed in intranet deployment. To summarize, the connectivity to AVES reusable-IP hosts is more easily disrupted by flooding attacks than that to regular IP hosts, however, AVES reusable-IP hosts are somewhat less vulnerable to other security exploits. Attackers also cannot cause waypoints to incorrectly relay traffic.

First and foremost, we acknowledge that AVES waypoints are no better at handling packet flooding type of denial of service attacks than any other network systems. The only method to prevent this is to traceback to the origin of the flooding and filter those packets out of the network. There has been some recent advances in this area [23]. Ingress filtering [5] also helps reduce the problem by disallowing address spoofed packets from entering the network. When the waypoints are flooded, reusable-IP networks will only have out-bound connectivity through NAT as in without AVES. In our implementation, we simply put some hard limits on resource consumptions to prevent overloading of each AVES component during a flooding attack. At a different level, to cope with aggressive users, the AVES-aware DNS server can potentially be extended to allocate the available session creation capacity more fairly by scheduling requests based on the initiators' and responders' identities. This way, an initiator or a responder (e.g. a popular web server) cannot occupy all resources and prevent other normal users from opening sessions. Currently, our implementation simply limits the peak rate at which sessions can be opened to each responder.

To address the second question, although AVES provides in-bound connectivity, it does not fully expose reusable-IP hosts and attacking them is somewhat more difficult. We have disabled the zone transfer [15] function of the AVES-aware DNS server to prevent malicious users from obtaining host names. In addition, to prevent scanning of host names, our implementation ignores and penalizes a requester that queries for host names that do not exist in our database. Without knowing any host name, the only opportunity for an attacker to connect to a reusable-IP host is to transmit packets to a waypoint during the time it is in a wait state. To lower the chance of this succeeding, our waypoint daemon monitors for in-coming packets with source

addresses that it has no state for while it is not in a wait state and reject all packets from these sources for 3 hours.

Finally, an attacker may hope to cause waypoints to misbehave by sending malicious packets to a waypoint while it is in a wait state. However, we have designed the wait state algorithm such that these malicious packets *cannot* cause a waypoint to mis-behave, they *cannot* prevent a legitimate initiator from connecting to the correct responder. The reason is that the wait state period is fixed and does not end simply because a malicious new initiator has arrived. The rejection algorithm is also conservatively designed to make sure all admission control violations are caught even in the presence of malicious packets.

6.3 Scalability

Because AVES is optimized for deployment, its scalability is a key concern. First, on the control path, since the AVES-aware DNS server can be replicated easily, DNS query processing should not present scalability problems. For intranet deployment, when local DNS server upgrades are possible, there is no protocol imposed limit on the rate at which sessions can be opened, and we have shown that a Linux PC waypoint can process thousands of requests per second. However, if the delayed binding technique is used, the rate at which the system can accept sessions is limited by the protocol. For our prototype system, 25 sessions can be accepted per second. Under such constraints, AVES should not be used to serve a busy web server. Note that this session acceptance rate limit *does not* reduce the connectivity achievable by the system as stated in Section 4.

On the data path, the scalability concern is whether the service provider's waypoints can handle the data traffic from initiators. Our experiments have demonstrated that our un-tuned implementation of AVES achieves a reasonable level of performance. With the advances in tera-bit class router technologies, we believe the data path operations can be performed at very high-speed. An alternative approach is to harness the resources of the NAT gateways of AVES subscribers, and use these NAT gateways as waypoints to relay subscribers' traffic. This way, the number of waypoints increases with the number of AVES subscribers, ensuring scalability. Although our software can be extended easily to support this service model, it introduces several new problems. Since waypoints are no longer owned by a trusted service provider, it is not clear what type of security protection can be achieved. Also, because waypoints can no longer be assumed to be always-on, maintaining the set of waypoints dynamically and providing fault tolerance are important problems to be addressed.

6.4 Potential Extensions

IPv6 Support – Our implementation currently does not support IPv6 header conversion, this is an important extension that is needed.

Coexisting with Ingress Filtering – Consider the example in Figure 4 again. In step 5, R is effectively spoofing IP_W . This is done for simplicity and performance reasons. Routers that implement ingress filtering [5] will drop such packets. AVES can easily be enhanced to work with ingress filtering by making R tunnel the packet to W , and let W forward the packet to A . The disadvantage is that the load on W is increased.

Coexisting with IPsec – To make NAT IPsec-compatible, RSIP [2] has recently been proposed in the IETF. In order for AVES to be compatible with IPsec, packet content must not be altered in transit. This can be achieved if the responder is made aware of the fact that it is being virtualized by a waypoint. This idea is in-spirit similar to that in RSIP. Using the example in Figure 4 again, the waypoint can generate the packet [$IP_W \rightarrow IP_R[IP_A \rightarrow IP_W]$] (step 2), R can forward the packet [$IP_R \rightarrow IP_B[IP_A \rightarrow IP_W]$] (step 3), and the responder itself can generate the packet [$IP_B \rightarrow IP_R[IP_W \rightarrow IP_A]$] (step 4). The reusable-IP responder now needs to be heavily modified, although there are some incentives to do so.

Connectivity for Non-IP Initiators – AVES is designed to solve the connectivity problem of cases (a) and (b) in Table 1. Since other cases are reducible to either case (a) or (b), AVES functions correctly in all cases. However, because AVES perceives all non-IP initiators belonging to the same non-IP network as a single IP initiator (since they are masked by their NAT or NAT-PT gateway), the connectivity provided by AVES to each individual non-IP initiator is correspondingly reduced. Precisely, with N IP addresses allocated for AVES waypoints, each non-IP network can simultaneously reach up to N non-IP responders. Although the connectivity is reduced, it is important to realize that this is perhaps the best one can achieve if the initiating non-IP network has no incentive to make any upgrade. If upgrading is acceptable, higher connectivity for these cases can be achieved by extending the NAT or NAT-PT gateways to implement a more sophisticated solution such as TRIAD [3] or IPNL [6]. A discussion on TRIAD and IPNL can be found in Section 7.

7 Related Work

In this section, we first review some well known partial work-arounds to cope with the lack of in-bound connectivity. Then we discuss a solution that is currently proposed in the IETF. Finally, we discuss other related work that are not directly addressing the in-bound connectivity problem.

A common work-around for the lack of in-bound connectivity is to forward a port number of the NAT gateway to a specific host inside the reusable-IP network. For example, in-coming traffic to port 23 (i.e., telnet) of the NAT gateway can be blindly redirected to port 23 of a particular reusable-IP host. With this transport layer work-around, although more than one reusable-IP host is reachable, no two reusable-IP hosts can offer the same service (e.g. no two

reusable-IP host can simultaneously support port 22 ssh login), thus unacceptable connectivity is provided. Tedious per application manual configurations are also required.

A related technique is to take advantage of a new type of DNS resource record proposed in [9], called the SRV resource record, which can specify the port number of a service. When both the service provider *and* the client support DNS SRV, the client can retrieve both the IP address of the host that is offering the service and the exact port number it should use to use the service. Suppose the port number binding can be dynamically assigned by a NAT gateway, then better in-bound connectivity to non-IP hosts can be achieved than port number forwarding. Unfortunately most applications and operating systems today do not support this feature.

Another work-around exists for UDP communication. Assume both the initiator and the responder are behind NAT gateways. The idea is to have both the initiator and the responder contact an IP server to exchange their NAT gateways' IP addresses, then both initiator and responder simultaneously send each other UDP packets with the same source and destination port numbers. Assuming the NAT gateways do not alter the source port numbers of these packets, bi-directional communication can be achieved. This work-around has been applied to some networked games [12]. Note that this scheme only works for UDP, requires a third party connection broker, and both parties must be actively involved, which is not suitable for client-server applications.

Another possibility is to insert a globally unique host name into packets so that a NAT gateway can dynamically determine the destination of a packet by looking up the host name. Host Identity Payload [17], proposed in the IETF, may be used for this purpose. Existing IP hosts or edge routers must however be modified to insert such host names into packets. With HTTP/1.1 [7], it is possible to embed the name of the destination in the HTTP header. This technique has been used to perform HTTP virtual hosting. This is however not a general solution for applications that are not based on HTTP.

Recently, a solution based on the SOCKS protocol has been proposed in the IETF [14]. The idea is that, when an application performs a DNS lookup for a responder, a “fake” IP address (e.g. 0.0.0.1) is returned to the application. When the application actually makes a socket call to communicate with the “fake” IP address, the SOCKS library on the initiator intercepts the call and connects to the SOCKS server on the responder’s NAT gateway. The DNS name of the responder is communicated to the SOCKS server, and the SOCKS server connects to the real responder. Data packets are then copied between the two spliced connections at the NAT gateway. The downside of this scheme is that existing IP hosts need to be upgraded. It is conceivable that the initiator-side’s SOCKS processing can be pushed to the initiator’s edge router; in that case,

existing edge routers need to be upgraded.

Next we discuss two on-going research projects that are closely related to AVES. In [3], Cheriton *et al.* propose a solution called TRIAD that can solve the IP address scarcity problem. TRIAD makes it possible to expand the Internet by arbitrarily connecting an unlimited number of IP network realms, each with its own 32-bit address space. TRIAD uses DNS names rather than addresses for global identification. During DNS name resolution, a sort of realm-to-realm source route is computed. A simple “shim” protocol header is added to every packet to carry this realm-to-realm source route to assist routing across multiple realms.

IPNL [6] is another recent proposal to provide an alternative to IPv6. IPNL also uses DNS names as global identifiers and allows multiple IP realms to be connected. However, rather than allowing IP realms to be connected arbitrarily as in TRIAD, IPNL allows IP realms to be organized hierarchically, with a single global “middle realm” and many smaller realms connected to the “middle realm”. This allows IPNL to have better routing efficiency compared to TRIAD. IPNL introduces two extra levels of optional headers to permit communication across realms. To communicate, the initial packet contains the DNS names of the source and the destination. As the packet traverses the realms, various addresses are resolved and stored in the packet headers. These addresses are then used for fast packet forwarding and the DNS names can be omitted.

While the goal of TRIAD and IPNL is to provide an alternative to IPv6, the goal of AVES is to maintain connectivity between today’s IP Internet and emerging networks of IPv6 and reusable-IP address spaces. In contrast to AVES, TRIAD and IPNL only allows hosts within realms running those respective protocols to communicate with each other. However, unlike TRIAD, AVES cannot route packets over an arbitrary number of IP networks, nor can AVES achieve the level of connectivity of TRIAD or IPNL. Nevertheless, we believe that maintaining connectivity between existing IP hosts and IPv6 or reusable-IP hosts is an important problem, therefore the trade-off is justified.

8 Summary

The main contribution we make in this paper is that we propose a waypoint service called AVES that can provide high connectivity from IP hosts to IPv6 or reusable-IP hosts without consuming many IP addresses or changing existing IP hosts and IP network routers. AVES is optimized for deployability and can be deployed easily as a 3rd-party network service. We have implemented and deployed a prototype system at CMU, and have received very positive feedbacks from our subscribers. Further information on AVES can be found at <http://www.cs.cmu.edu/~eugeneng/research/aves/>.

References

- [1] S. Bellovin. Firewall friendly FTP, February 1994. RFC-1579.
- [2] M. Borella, J. Lo, D. Grabelsky, and G. Montenegro. Realm Specific IP: A framework, July 2000. Internet Draft, draft-ietf-nat-rsip-framework-05.txt.
- [3] D. R. Cheriton and M. Gritter. TRIAD: A new next generation Internet architecture, March 2000. <http://www-dsg.stanford.edu/triad/triad.ps.gz>.
- [4] S. Deering and R. Hinden. Internet Protocol, version 6 (IPv6) specification, December 1998. RFC-2460.
- [5] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing, May 2000. RFC-2827.
- [6] P. Francis. IPNL architecture and protocol description, May 2000. Available from the author.
- [7] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1, June 1999. RFC-2616.
- [8] R. Gilligan and E. Nordmark. Transition mechanisms for IPv6 hosts and routers, April 1996. RFC-2893.
- [9] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV), February 2000. RFC-2782.
- [10] M. Holdrege and P. Srisuresh. Protocol complications with the IP network address translator, January 2001. RFC-3027.
- [11] Internet Software Consortium. Berkeley Internet Name Domain (BIND) version 8.2.3. <http://www.isc.org/products/BIND/>.
- [12] D. Kegel. NAT and peer-to-peer networking. <http://www.alumni.caltech.edu/~dank/peer-nat.html>.
- [13] S. Kent and R. Atkinson. Security architecture for the Internet Protocol, November 1998. RFC-2401.
- [14] H. Kitamura. A SOCKS-based IPv6/IPv4 gateway mechanism, March 2001. Internet Draft, draft-ietf-ngtrans-socks-gateway-06.txt.
- [15] P. Mockapetris. Domain names - concepts and facilities, November 1987. RFC-1034.
- [16] J. Mogul and S. Deering. Path MTU discovery, November 1990. RFC-1191.
- [17] R. Moskowitz. Host Identity Payload, February 2001. Internet Draft, draft-moskowitz-hip-arch-02.txt.
- [18] E. Nordmark. Stateless IP/ICMP translation algorithm (SIIT), February 2000. RFC-2765.
- [19] J. Postel. Internet Control Message Protocol, September 1981. RFC-792.
- [20] J. Postel. Internet Protocol, September 1981. RFC-791.
- [21] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address allocation for private internets, February 1996. RFC-1918.
- [22] R. Rivest. The MD5 message-digest algorithm, April 1992. RFC-1321.
- [23] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for IP traceback. In *Proceedings of ACM SIGCOMM’00*, Stockholm, Sweden, Aug. 2000.
- [24] D. Senie. NAT friendly application design guidelines, March 2001. Internet Draft, draft-ietf-nat-app-guide-05.txt.
- [25] P. Srisuresh and K. Egevang. Traditional IP network address translator (Traditional NAT), January 2001. RFC-3022.
- [26] P. Srisuresh, G. Tsirtsis, P. Akkiraju, and A. Heffernan. DNS extensions to network address translators (DNS_ALG), September 1999. RFC-2694.
- [27] G. Tsirtsis and P. Srisuresh. Network address translation - protocol translation (NAT-PT), February 2000. RFC-2766.

Flexible Control of Parallelism in a Multiprocessor PC Router

Benjie Chen

Robert Morris

*Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts, 02139
{benjie,rtm}@lcs.mit.edu*

Abstract

SMP Click is a software router that provides both flexibility and high performance on stock multiprocessor PC hardware. It achieves high performance using device, buffer, and queue management techniques optimized for multiprocessor routing. It allows vendors or network administrators to configure the router in a way that indicates parallelizable packet processing tasks, and adaptively load-balances those tasks across the available CPUs.

SMP Click's absolute performance is high: it can forward 494,000 64-byte IP packets per second on a 2-CPU 500 MHz Intel Xeon machine, compared to 302,000 packets per second for uniprocessor Click. SMP Click also scales well for CPU intensive tasks: 4-CPU SMP Click can encrypt and forward 87,000 64-byte packets per second using IPSec 3DES, compared to 23,000 packets per second for uniprocessor Click.

1 Introduction

High performance routers have traditionally forwarded packets using special purpose hardware. However, many routers are expected to perform packet processing tasks whose complexity and variety are best suited to software. These tasks include encrypting virtual private network tunnels, network address translation, and sophisticated packet queuing and scheduling disciplines. These tasks are likely to be too expensive for a single CPU at high line rates. Many routers already include multiple CPUs to exploit parallelism among independent network links [26], and the advent of routers with multiple tightly-coupled CPUs per link seems near [6, 10, 11]. This paper describes and analyses techniques to extract good performance from multiprocessor PC routers with a variety of packet processing workloads.

In order to increase performance, a multiprocessor router must find and exploit operations that can be carried out si-

multaneously. Potential parallelism arises naturally in loaded routers as multiple packets queue up at inputs waiting to be processed. However, good performance demands some care in the way that packet processing tasks are divided among multiple CPUs. Any single packet should be processed by as few distinct CPUs as possible, to avoid cache conflicts. Each mutable data structure, such as a queue or device driver state record, should be touched by as few distinct CPUs as possible to avoid locking costs and cache conflicts. Similarly, if the router keeps mutable state for a flow of packets, processing for all packets of that flow should be done on the same CPU. Finally, the number and costs of the tasks should permit balancing the processing load and avoiding idle CPUs. The best way to split up a router's work among the CPUs depends on the router's packet processing and on traffic patterns.

An ideal multiprocessor router would allow configuration of its parallelization strategy in conjunction with configuration of its packet processing behavior. This paper describes a system, SMP Click, for doing so. SMP Click is derived from the Click [14] modular router. Click routers are configured with a language that declares packet processing modules and the connections among them. SMP Click provides automatic parallel execution of Click configurations, using hints from the configuration structure to guide the parallelization. Thus a router vendor or network administrator can easily tailor the way that a multiprocessor router parallelizes its packet processing tasks in order to maximize performance. This paper describes how SMP Click works and how it supports configurable parallelization.

This paper contributes the following lessons about SMP router design. First, no one approach to parallelization works well for all router configurations. Second, parallelization techniques can be effectively expressed at the level of router configurations, and such configurations can be restructured to increase performance. Finally, significant parallelism can often be found even in untuned configurations.

The next section presents an overview of Click and describes the example configurations used in the rest of the paper. Section 3 describes SMP Click's design goals. Section 4 details the challenges faced in its implementation along with

This research was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Space and Naval Warfare Systems Center, San Diego, under contract N66001-00-1-8933.

their solutions. Section 5 analyzes SMP Click’s performance with configurations not tailored to multiprocessors. Section 6 presents several ways that SMP Click allows control over parallelism, along with the resulting performance improvements. Section 7 describes related work, and Section 8 concludes the paper.

2 Click

This section introduces the Click router toolkit. A complete description is available in Kohler’s thesis [13]; the element glossary in its Appendix A may be particularly helpful.

Click routers are built from modules called *elements*. Elements process packets; they control every aspect of router packet processing. Router configurations are directed graphs with elements as the vertices. The edges, called *connections*, represent possible paths that packets may travel. Each element belongs to an *element class* that determines the element’s behavior. An element’s class specifies which code to execute when the element processes a packet. Inside a running router, elements are represented as C++ objects and connections are pointers to elements. A packet transfer from one element to the next is implemented with a single virtual function call.

Each element also has *input* and *output ports*, which serve as the endpoints for packet transfers. Every connection leads from an output port on one element to an input port on another. Only ports of the same kind can be connected together. For example, a push port cannot be connected with a pull port. An element can have zero or more of each kind of port. Different ports can have different semantics; for example, the second output port is often reserved for erroneous packets.

Click supports two packet transfer mechanisms, called *push* and *pull processing*. In push processing, a packet is generated at a source and passed downstream to its destination. In pull processing, the destination element picks one of its input ports and asks that source element to return a packet. The source element returns a packet or a null pointer (which indicates that no packet is available). Here, the destination element is in control—the dual of push processing.

Every queue in a Click configuration is explicit. Thus, a configuration designer can control where queuing takes place by deciding where to place *Queue* elements. This enables valuable configurations like a single queue feeding multiple interfaces. It also simplifies and speeds up packet transfer between elements, since there is no queuing cost.

Click provides a language for describing router configurations. This language declaratively specifies how elements should be connected together. To configure a router, the user creates a Click-language file and passes it to the system. The system parses the file, creates the corresponding router, tries to initialize it, and, if initialization is successful, installs it and starts routing packets with it.

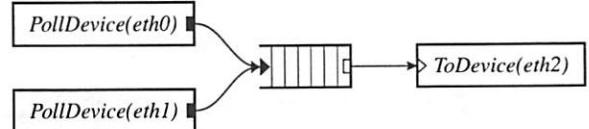


Figure 1: A simple Click configuration.

Figure 1 shows a simple Click configuration. In our configuration diagrams, black ports are push and white ports are pull; agnostic ports, which can connect to either push or pull ports, are shown as push or pull ports with a double outline. This configuration reads packets from network interfaces named *eth0* and *eth1*, appends them to a queue, and transmits them out interface *eth2*. The *PollDevices* initiate pushes along the paths to the queue as packets arrive. The *ToDevice* initiates pulls from the queue as the device hardware completes previous packet transmissions.

2.1 An IP Router

Figure 2 shows a basic 2-interface IP router configuration. Detailed knowledge of this configuration is not required to understand this paper; it’s included to give a feel for the level at which one configures a Click router.

The high-level flow of packets through Figure 2 is as follows. Each *PollDevice* element reads packets from an input device. The *Classifier* separates ARP queries and responses from incoming IP packets. *Paint* annotates each packet with the index of the interface it arrived on, for later use in generating redirects. *Strip* removes the 14-byte Ethernet header, leaving just an IP packet. *CheckIPHeader* verifies that the IP checksum and length fields are valid. *GetIPAddress* extracts the packet’s destination address from the IP header. *LookupIPRoute* separates the packets according to which output interface they should be sent to; it also separates packets addressed to the router itself. The elements before the *LookupIPRoute* perform per-interface input processing; the elements after the *LookupIPRoute* perform per-interface output processing.

The first stage in output processing is to drop any packet sent to a broadcast Ethernet address, since forwarding it would not be legal. *CheckPaint* detects a packet forwarded out the same interface on which it arrived, and arranges to generate an ICMP Redirect. *IPGWOptions* processes hop-by-hop IP header options. *FixIPSrc* rewrites the source address of any packet generated by the router itself to be the address of the outgoing interface. *DecIPTTL* checks and decrements the TTL field, and *IPFragmeneter* fragments large packets. *ARPQuerier* finds the Ethernet address associated with the next hop and prepends an Ethernet header; this may involve setting aside the packet while sending out an ARP query. Finally, the push path ends by depositing the packet in a *Queue*. *ToDevice* pulls packets out of the queue whenever the output device is ready to send.

2.2 Configuration-level Parallelism

Control (i.e. a CPU thread) can enter a Click configuration at one of only a few *schedulable* elements: at a *PollDevice* element, to check device hardware for new input packets and start push processing; at a *ToDevice*, to initiate a pull for the next available output packet and send it to device output hardware; and at a *PullToPush* element which initiates a pull through its input and pushes any resulting packet to its output. Once a CPU thread starts pull or push processing for a packet at a schedulable element, that thread must carry the packet through the configuration until it reaches a *Queue*, a *ToDevice*, or some other element that discards or otherwise disposes of the packet. For convenience, let a *push path* be a sequence of elements that starts with a schedulable push element, such as *PollDevice*, and ends with a *Queue*, and let a *pull path* be a sequence of elements that starts with a *Queue* and ends with a schedulable pull element, such as *ToDevice*.

These constraints on control flow mean that a Click configuration conveys a good deal of information about potential parallelism. CPUs executing completely disjoint paths will not interfere with each other at all. CPUs carrying packets along the same path may interfere with each other, though parallelism may still be available if the path contains multiple expensive elements.

A common situation arises when paths from a number of *PollDevice* elements converge on a *Queue*, which in turn feeds a *ToDevice*. The push paths from the *PollDevices* are mostly disjoint, conflicting only at the last element *Queue*, so each can be profitably executed by a separate CPU. In contrast, the pull path from the *Queue* to the *ToDevice* is usually short and would cause contention if executed on multiple CPUs; in fact, SMP Click never executes any schedulable element on more than one CPU concurrently.

3 Design Goals

The most obvious design goal of SMP Click is to run Click configurations on multiprocessor PC hardware. In order for it to be useful, however, it must achieve a number of related goals:

- SMP Click users should not need to think about synchronization when writing configurations. Configurations that work on a uniprocessor should also work correctly on multiprocessors.
- SMP Click should improve the performance of even naive configurations, so that no special skills are required to take some advantage of it.
- It should be easy to rewrite Click configurations to expose parallelism and thus increase performance.

The second goal is reasonable because most Click configurations inherently allow for some parallelism. Any config-

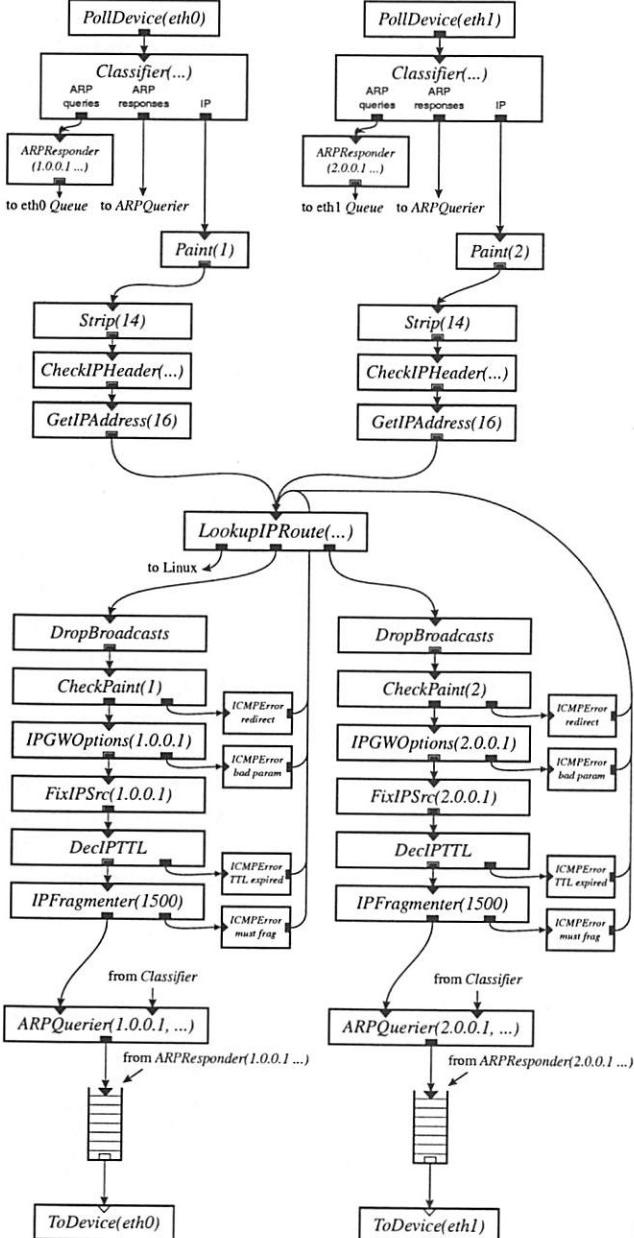


Figure 2: An IP router configuration with two network interfaces. This router implements RFC 1812 [2].

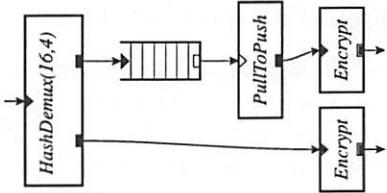


Figure 3: A configuration fragment that splits packets into two flows based on their destination IP addresses. Because the new *PullToPush* element can be scheduled separately, two threads can perform the expensive encryption operation.

uration with multiple network interfaces can read from and write to the different interfaces in parallel. Checking incoming packets for correctness and even lookup of destinations in routing tables can also often proceed in parallel for packets from different interfaces. For example, almost all of Figure 2 can proceed in parallel for packets arriving from different interfaces; contention first occurs either in the mutable ARP table in *ARPQuerier* or in the queues.

In support of the last goal, SMP Click allows users to perform a variety of configuration transformations that might increase parallelism, including the following:

- **Pipeline paths.** If a single push path contains multiple expensive elements, it may be advantageous to break up the path to allow pipelining on multiple CPUs. This can be done by inserting a *Queue* and a *PullToPush* element in the path. A *PullToPush* can be scheduled on its own CPU, and thus can initiate the processing for one pipeline stage.
- **Split into separate flows.** If a configuration contains only one expensive element, the push path cannot be pipelined. Sometimes the expensive element itself can be replicated and executed in parallel on different packets, however. Incoming packets from the same flow usually need to be directed to the same replica, both to preserve order within flows and because the replicated element may maintain per-flow state. Figure 3 shows a configuration fragment that splits flows. In this fragment, the *HashDemux* element demultiplexes incoming packets based on their destination IP addresses. The new *PullToPush* element becomes an additional point where a thread may enter the configuration to perform expensive packet processing, such as encryption.

See Section 6 for examples of some of these transformations and their effects on performance.

4 Implementation

Uniprocessor Click, the predecessor to the work described here, runs in a single thread inside the Linux kernel. It sched-

ules work by maintaining a work list of elements that want CPU time. These elements are typically of types *PollDevice*, *ToDevice*, and *PullToPush*. Since these elements poll for the availability (or departure) of packets, and Click uses no interrupts, they must be called periodically. All pushes and pulls are initiated by elements on the worklist.

SMP Click retains much of the structure of uniprocessor Click, but involves changes in a number of areas. These include scheduling the worklist on multiple CPUs, synchronization to protect mutable data in re-entrant elements, and special handling of devices, buffer free lists, and queues to enhance parallelism.

4.1 CPU Scheduling

When it first starts, SMP Click creates a separate thread for each processor. Each thread runs schedulable elements from a private worklist in round-robin order, occasionally yielding control to Linux so user processes can make progress. This approach differs from most software routers built on top of traditional operating systems in that packet processing is not driven by packet arrival interrupts, hence device handling cannot starve packet forwarding [17].

Each thread has a private worklist in order to avoid the expensive synchronization operations associated with centralized worklists [1] and to allow processor affinity scheduling. Load balancing among these private worklists, however, is difficult to achieve for three reasons. One, Click never interrupts an element while it is processing a packet, so timeslicing is not possible. Two, since elements take different time to execute, merely balancing the number of elements on each worklist is not adequate. Three, because Click is not event driven, an idle element cannot remove itself from a worklist and rejoin the list later on when it is ready to process a packet. Consequently, most schedulable elements remain on the worklist even if they rarely have work to do. This means SMP Click cannot use work-stealing algorithms [1, 5] that steal work from other worklists when the local worklist empties.

SMP Click offers two solutions for load balancing. It provides an adaptive load balancing algorithm that schedules elements onto different CPUs, providing good load balance. It also allows ambitious users to statically schedule elements based on SMP Click’s performance measurement tools. We describe both approaches below.

4.1.1 Adaptive CPU Scheduling

When an SMP Click router starts, one worklist contains all schedulable elements. Click maintains, for each schedulable element e , the average cost of that element, C_e . If adaptive CPU scheduling is used, a global scheduler rebalances the assignment of elements to worklists periodically. The scheduler sorts the schedulable elements in decreasing order based on

C_e . It then iterates through the sorted list, assigning each element to the worklist with the least amount of total work so far.

The cost of an element is the average number of cycles consumed by the push or pull processing initiated by this element each time it is called. To obtain this number, SMP Click periodically samples the number of cycles consumed by the element when it is called. This sampling technique does not introduce any noticeable performance overhead.

The CPU scheduling mechanism described here provides three benefits. It balances useful work among the CPUs to increase parallelism, it avoids contention over a single work list, and it encourages affinity between particular tasks and CPUs to reduce cache misses.

4.1.2 Static CPU Scheduling

Adaptive load balancing may not result in the best routing performance due to its ignorance of cache miss costs. If two elements that process the same packets (e.g. a *PollDevice* element and the *ToDevice* element that it sends packets to) are scheduled onto different CPUs, the cost of processing each packet increases due to cache misses. Thus, even if a balanced load is achieved, the router may still perform worse than when these two elements are scheduled onto the same CPU.

Automatically instrumenting the packet processing code to detect cache misses would involve reading hardware performance counters, a costly operation. On the other hand, with some knowledge about the costs of different paths, a user can easily specify a good scheduling assignment for most configurations. For example, a four-interface IP router has eight schedulable elements: four *PollDevice* elements and four *ToDevice* elements. Because the path initiated by the *PollDevice* is more expensive than the path initiated by the *ToDevice*, a good scheduling assignment on four CPUs would schedule one *PollDevice* and one *ToDevice* on each CPU. Furthermore, to reduce the cost of cache misses, *PollDevice* and *ToDevice* elements that operate on the same interface should not be scheduled together, since they never process the same packets.

Static scheduling can be specified in the form of a list of assignments of schedulable elements to CPUs. In addition, Click can be configured to measure and report the execution time of packet processing paths.

4.2 Synchronization

Any element instance in SMP Click might be executed simultaneously on multiple CPUs, so every element must protect its mutable data structures. The details are private to the implementation of each element type, since elements don't use each other's data. A number of different approaches prove useful.

Many elements have no mutable state, and thus require no

special synchronization. A typical example is the *Strip* element, which simply removes bytes from the head of a packet.

Some elements have state composed of just a counter. If the counter is rarely incremented, as in the case of an error counter, it can be updated with hardware atomic increment instructions. A typical example is the *CheckIPHeader* element, which maintains a count of invalid packets.

Some mutable element state can be replicated per processor, so that it is never shared. For example, the IP routing table lookup element keeps a private per-CPU cache of recently used routes, rather than a single shared cache.

Some elements protect their state with spin-locks, implemented with the x86 *xchgw* atomic exchange instruction. If a CPU acquires a lock that was last held by the same CPU, the *xchgw* executes quickly out of that CPU's cache; otherwise the *xchgw* involves a slow off-chip bus transaction. Thus, for data which is only occasionally written, SMP Click uses read/write locking in which each CPU has its own read lock, and a writer has to acquire all the read locks. *ARPQuerier* and *IPRewriter* use this technique to protect their tables.

An element instance that appears on the work list is executed by at most one CPU at a time. Thus *PollDevice* and *ToDevice* elements need not take special pains to prevent more than one CPU from communicating with the same device hardware.

Device handling, the buffer free list, and queues need special attention for high performance, detailed in subsequent sections.

4.3 Device Handling

Click device drivers use polling rather than interrupts in order to avoid interrupt overhead. An alternate approach might have been to use the “interrupt coalescing” scheme supported by the Intel Pro/1000 F gigabit Ethernet cards we used, which lowers interrupt overhead by imposing a minimum delay between successive interrupts. The correct minimum delay parameter turns out to depend on the time required to completely process all packets that arrive on all interfaces between interrupts, which proved too difficult to predict. Another reason to prefer polling is that it eliminates the expense of synchronization between threads and interrupt routines.

To maximize parallelism, SMP Click device drivers completely separate transmit and receive data structures. For example, the transmit routines are responsible for freeing transmitted packets, and the receive routines are responsible for giving the device fresh empty buffers.

Polling a device that has no packets waiting needs to be very fast. In practice this means that the device's DMA descriptors should reside in host memory (not in device memory), and that the driver should be able to discover new packets just by looking at the descriptors. If the device is idle, the CPU will have already cached these descriptors, and checking them will be fast.

It is also important that the driver and the device should never explicitly synchronize or directly communicate. The host should not read or write on-device registers to exchange information about new packets or free buffers; instead all such communication should take place indirectly through DMA descriptor contents. The Intel 21140 [7] is a good example of such a design. Unfortunately, the Intel Pro/1000 devices used for this paper’s experimental results require the driver to write device registers to announce the addition of buffers to DMA descriptors. The SMP Click drivers reduce this overhead by batching such additions.

4.4 Buffer Management

Packet buffers in a router usually go through a repeating life-cycle: they are allocated by a device driver, filled with incoming data by a device, processed by the router, transmitted, and then freed. SMP Click takes advantage of this regularity in a number of ways.

SMP Click uses Linux’s `sk_buffs`, which consist of a descriptive structure (containing lengths etc.) and a separate data buffer. `sk_buff` data is contiguous, which makes it easy to manipulate; lists of sub-buffers (as in BSD `mbufs` [16]) are mainly useful for host protocols in which headers and payload may be stored separately.

The `sk_buff` allocator in Linux 2.2.18 is expensive. Allocating (or freeing) an `sk_buff` requires two locking operations, since the structure and the data buffer are allocated separately. Since Linux has only one free list, it is likely that an `sk_buff` freed on one CPU will be allocated on another, causing needless cache misses.

SMP Click avoids these costs by handling `sk_buff` allocation itself. It maintains a separate free list for each CPU, implemented as a circular array. Each CPU frees only onto its own free list, so freeing never requires locking. When a CPU needs to allocate a packet, it tries to do so from its own free list to benefit from the possibility that the buffer is already cached. If its free list is empty, it allocates a packet from another CPU’s free list. The other list is chosen in a way that makes it likely that each CPU has at most one other CPU allocating from its list. Since it is common for some CPUs to be dedicated to receiving only, and some to transmitting only, this matches up producers and consumers of free packets. Producers never need to lock, and consumers usually acquire a lock that they were the last to hold, which is fast.

As an additional optimization, both allocation and freeing are batched, decreasing free list manipulation costs. Batching is possible since only device drivers ever free or allocate, and they can arrange to defer such actions until they can perform them on many device DMA descriptors at once.

4.5 Queues

Queue elements are the primary points at which packets move

from one CPU to another, so accesses to *Queue* data structures and enqueued packets are likely to cause cache misses. In addition, multiple threads may enqueue and dequeue from a *Queue*, so it must protect its data structures. In many common cases, however, these costs can be eliminated.

Most *Queues* are used to feed output devices. Two *ToDevices* can share a single *Queue*, if they are feeding parallel links to another router, but this is not a common situation. SMP Click automatically eliminates locking for dequeues in the usual case in which only one *ToDevice* pulls from a *Queue*. This is possible because a *Queue* is implemented with a circular array of buffer pointers, and the enqueue and dequeue operations modify different pointers into that array.

Most *Queues* are fed by multiple *PollDevices*, and must be prepared for concurrent enqueues. SMP Click enqueues with an atomic compare and swap instruction to avoid some locking overhead.

4.6 Batching and Prefetching

SMP Click processes packets in batches to reduce cache coherency misses, to amortize the cost of locks over multiple packets, and to allow effective use of memory prefetch instructions. Batching is implemented by *PollDevice*, *Queue*, and *ToDevice* elements; other elements and all inter-element communication are one packet at a time.

PollDevice dequeues up to eight packets from the device DMA queue at a time, then sends them one by one down the push path. Batching the device dequeues allows the driver to usefully prefetch DMA descriptors and packet contents, which are not in the CPU cache since they were last written by device DMA. Batching also allows the driver to allocate new receive buffers in batches, amortizing the overhead of locking the free list.

ToDevice tries to pull multiple packets from its upstream *Queue* each time it is called by the scheduler. It enqueues these packets onto the transmit DMA ring. After the entire batch has been enqueued, *ToDevice* notifies the device of the new packets. Batching allows the device driver to amortize the cost of this notification over many packets. In addition, *ToDevice* frees transmitted packets in groups.

Multiple packets must be enqueued in a *Queue* in order for *ToDevice* batching to be effective. To ensure that the *ToDevice* can pull several packets at a time from the *Queue*, the dequeue code pretends that the queue is empty until either eight packets have been enqueued, or a short time has elapsed. This also allows the enqueueing CPU to keep the queue data structures in its cache while it enqueues a few packets; otherwise the enqueueing and dequeuing CPUs would fight over those cache entries.

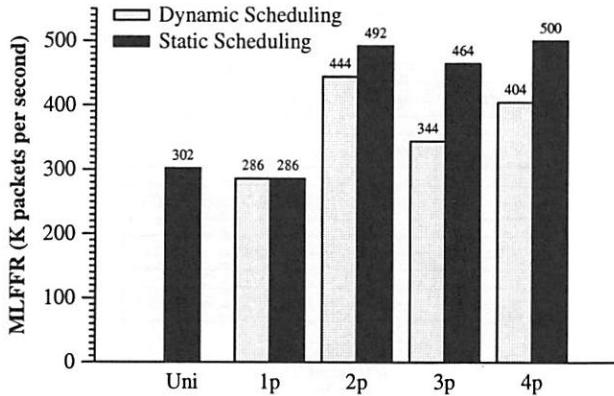


Figure 4: Maximum forwarding rates of the IP configuration. The Uni column refers to uniprocessor Click running on a uniprocessor Linux kernel. The xp columns refer to SMP Click with x CPUs. SMP Click is not able to take advantage of more than two CPUs with this configuration.

5 Performance of Naive Routers

This section examines SMP Click’s performance with some configurations originally designed for use on uniprocessor Click; the results reflect on SMP Click’s goal of increased performance for untuned configurations.

5.1 Experimental Setup

The experimental setup consists of five Intel PCs running Linux 2.2.18. One of the PCs acts as a router, with a separate full-duplex point-to-point gigabit Ethernet link to each of the other four “host” PCs. The host PCs send packets into the router to be forwarded to the other host PCs. The router’s IP routing table contains just the entries required for the four hosts.

The router is a Dell PowerEdge 6300, with four 500 MHz Intel Pentium III Xeon CPUs, an Intel 450NX chipset motherboard, and 1GB of RAM. The hosts have dual 800 MHz Pentium III CPUs, ServerWorks LE chipsets, and 256MB of RAM. All the network devices are Intel Pro/1000 F gigabit Ethernet cards, connected to the motherboards with 64 bit 66 MHz PCI.

All experiments use 64-byte IP packets. Each packet includes Ethernet, IP, UDP or TCP headers, a small payload, and the 4-byte Ethernet CRC. When the 64-bit preamble and 96-bit inter-frame gap are added, a gigabit Ethernet link can potentially carry up to 1,488,000 such packets per second. Each host can send up to 876,000 packets per second, using software that closely controls the send rate. The hosts can also receive reliably at the same rate.

Task	uni	1p	2p	3p	4p
Recv	0.25	0.25	0.29	0.38	0.62
Alloc Buf	0.10	0.11	0.13	0.55	0.21
Refill	0.10	0.10	0.11	0.11	0.12
Push	1.80	1.83	2.19	3.03	3.60
Pull	0.18	0.26	0.33	0.45	0.77
Xmit	0.46	0.56	0.67	0.92	1.28
Clean	0.26	0.26	0.33	0.47	0.70
Free Buf	0.13	0.13	0.13	0.29	0.23
Total	3.28	3.50	4.18	6.20	7.53

Table 1: Cost of forwarding a packet in microseconds, broken down by function. Recv refers to reading DMA descriptors, Alloc Buf to allocation of new buffers, Refill to placing new buffers in DMA descriptors, Push to push processing (including enqueue), Pull to dequeue from the Queue, Xmit to placing packets on the transmit DMA ring, Clean to removing transmitted packets from the ring, and Free Buf to freeing them.

5.2 IP Performance

Figure 4 shows SMP Click’s performance when forwarding IP packets with a four-interface version of the configuration in Figure 2. In these experiments, each host sends IP packets to the other three hosts for 60 seconds. The y-axis represents the maximum loss-free forwarding rate (MLFFR). The x-axis represents five experimental scenarios: uniprocessor Click on uniprocessor Linux, and SMP Click on SMP Linux with one to four CPUs. We ran two experiments with each scenario, one with adaptive load balancing, one with static scheduling. In the static scheduling experiments, each *PollDevice* was scheduled on the same CPU as one of the three *ToDevices* that it forwards packets to. With two CPUs, two *PollDevices* and two *ToDevices* are scheduled on each CPU. With three CPUs, two *ToDevices* and a *PollDevice* are scheduled on two of the CPUs, with the remaining two *PollDevices* on the third. With four CPUs, each CPU runs one *PollDevice* and one *ToDevice*.

Table 1 helps explain these results by showing the CPU-time costs of forwarding a packet, measured with Intel Pentium cycle counters. The actual forwarding rates are close to those implied by the CPU time measurements. For example, Table 1 shows that it takes 4.18 microseconds of CPU time to forward a packet on a 2-CPU router, implying that each CPU should be able to forward 239,234 packets per second, and that the two CPUs together should be able to forward 478,468 packets per second; this is close to the actual rate of 444,000 to 492,000 packets per second measured in Figure 4.

As the number of CPUs increases, the per-packet CPU time also increases. This is because synchronization and cache misses impose costs that increase with the number of CPUs. The largest increase occurs for push processing, which includes enqueueing on *Queues*; the reason is that more CPUs

Task	uni	1p	2p	3p	4p
Recv	0.5	0.5	0.5	0.5	0.5
Alloc Buf	0.0	0.0	0.0	0.4	0.0
Refill	0.2	0.2	0.2	0.2	0.2
Push	2.0	2.0	3.0	3.7	4.7
Pull	0.0	0.0	0.5	1.1	1.4
Xmit	0.5	0.5	0.8	1.1	1.3
Clean	0.5	0.5	1.1	1.7	1.7
Free Buf	0.0	0.0	0.1	0.6	0.2
Total	3.7	3.7	6.2	9.3	10.0

Table 2: Number of system bus operations per IP packet forwarded; these correspond to cache misses and transfers of locks between CPUs.

cause more contention between enqueueing and dequeuing. The other expensive increases occur when allocating and freeing buffers, especially for the three CPU case. With three CPUs, one of the CPUs does not have any *ToDevice* elements. Thus it must allocate buffers from another CPU’s free list, resulting in cache misses on the buffer data structure and on the synchronized free list. Finally, Pull, Xmit, and Clean operations operate on dequeued buffers. As the number of CPU increases, they are more likely to have been touched last by another CPU.

Table 2 illustrates contention between CPUs by showing the number of system bus operations per packet. A bus operation is caused by an L2 cache miss, a write of shared/cached data, or an acquisition of a lock by a CPU other than the CPU that held it last. The cost of sharing *Queues* between enqueueing and dequeuing CPUs is evident in the increased number of bus operations on the Push and Pull lines as the number of CPUs increases.

The reason that dynamic load balancing does not work as well as static scheduling in Figure 4 is that the dynamic scheduler sometimes puts the *PollDevice* and *ToDevice* of the same interface on the same CPU. This misses opportunities to do both push and pull processing for some packets on the same CPU.

Some of the push and pull costs in Table 1 are due to IP processing. By using a much simpler configuration, essentially consisting of just device drivers, the potential performance of the underlying machine can be estimated. With a configuration that directly passes packets from each input interface through a *Queue* to a statically paired output with no intervening processing, SMP Click can forward 528,000 packets per second on one CPU and 566,000 packets per second on two or four CPUs. Each packet is processed entirely by a single CPU.

The above tests emphasize per-packet overheads, since they use small packets. With 200 byte UDP packets, the IP router has a MLFFR of 240,000 packets per second on four CPUs, or 366 megabits per second. With 1024 byte UDP

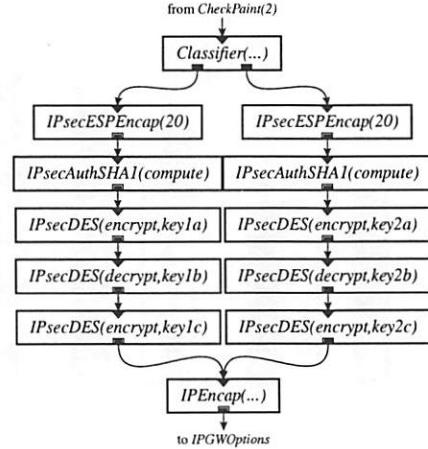


Figure 5: An IPSec VPN tunnel encryption configuration.

packets, the MLFFR is 90,000 packets per second on four CPUs, or 703 megabits per second.

This section shows that, with the Click SMP architecture, the benefits of parallelizing IP processing outweigh the costs of synchronization and data movement between CPUs by a relatively small margin. The next section demonstrates that parallelization is much more attractive for more compute-intensive packet processing.

5.3 Virtual Private Network Gateway

Figures 5 and 6 present Click configuration fragments that implement part of IPSec [12]. Inserting these elements into the IP router in Figure 2 produces a Virtual Private Network (VPN) gateway. The intent is that Figure 5 be inserted into the output processing of the router’s link to the outside world, to authenticate, encrypt, and encapsulate packets sent along a VPN tunnel to a similar remote router. Figure 6 performs the inverse operations for packets arriving from the interface to the outside world. The configurations shown use SHA-1 for authentication and 3DES for encryption.

For space reasons, Figure 5 and 6 show configurations with only two VPN tunnels, while our performance evaluation uses eight. These tunnels are established statically, using *IPClassifier* elements as the input and output security association databases.

The traffic used to test the VPN configuration is generated as follows. Two hosts are “internal” hosts; the other two are external hosts. Each internal host generates eight streams of ordinary 64-byte IP packets, four to one external host, and four to another. The VPN router authenticates, encrypts, encapsulates, and forwards these packets. Each external host generates eight streams of encapsulated, authenticated, and encrypted packets, four to each internal host. The VPN router unencapsulates, decrypts, authenticates, and forwards these packets as well.

Figure 7 shows the forwarding performance of the VPN

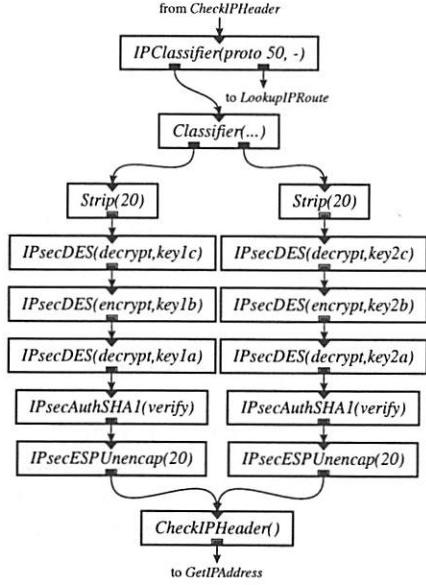


Figure 6: An IPSec VPN tunnel decryption configuration.

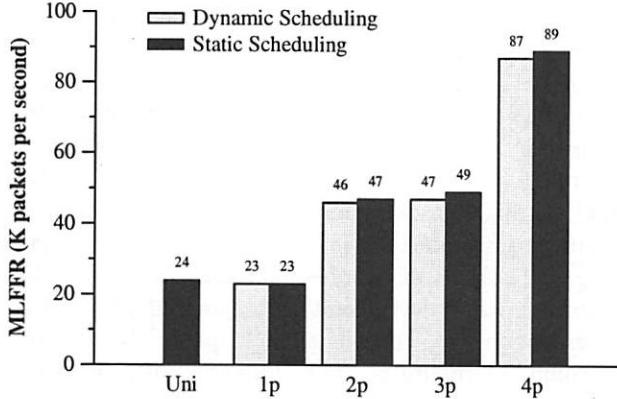


Figure 7: Performance of the IPSec VPN configurations. The forwarding rate increases linearly with the number of CPUs.

router in 60 second experiments. The static CPU scheduling experiments used the same scheduling assignment as the IP router. The VPN scales better than the IP router because computation performed on each packet is more expensive and dominates the overhead of cache misses and synchronization. For the same reason, adaptive load balancing works as well as static scheduling.

The reason the performance is no higher with three CPUs than with two is that one of the three CPUs has to handle encryption and decryption for two input interfaces. That CPU runs out of cycles, and starts dropping packets, when the other two CPUs are still only half utilized. This limits the loss free forwarding rate of the whole router.

As a crude comparison, a \$10,000 commercial VPN box with hardware assisted 3DES encryption was recently rated at 27 Mbps for 64 byte packets (i.e. 55 Kpps) [21].

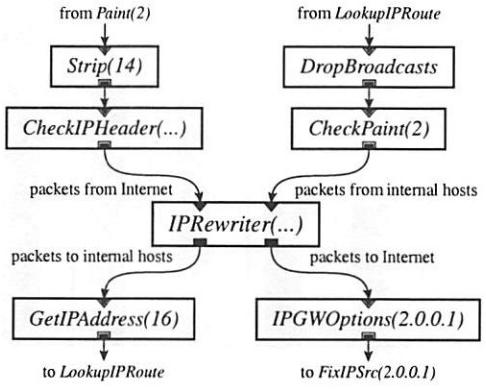


Figure 8: A Network Address Translator (NAT) configuration.

5.4 NAT Performance

Figure 8 shows a configuration fragment that, when inserted into the IP router configuration, implements a network address translator [9]. The router sends packets from “inside” hosts to external destinations to the upper-right input port of the *IPRewriter* [15]. The *IPRewriter* changes these packets’ source IP addresses to an externally visible address, rewrites the source TCP port numbers, and emits them out the lower-right output port; from there they are transmitted to the external Internet. The *IPRewriter* dynamically maintains tables that allow it to map all the packets of each connection in a consistent way, and allow it to associate incoming packets from the outside world with the relevant connection. Before the routing table lookup, each incoming packet enters the *IPRewriter* via the upper-left input port. The *IPRewriter* changes the destination IP address and TCP port number to that of the original connection, and sends the packet out on the lower-left output port. The correct destination route is then determined, using the updated destination IP address, by *LookupIPRoute*.

An *IPRewriter* handles mappings for a single externally visible IP address. It gives each connection its own externally visible port number. The *IPRewriter* remembers which connections have sent TCP FIN (connection close) messages, and deletes any such connection from its tables after 30 seconds. *IPRewriter* does this deletion incrementally: each time it sees a TCP SYN (connection setup) message, it checks to see if the oldest closed connection is 30 seconds old.

This deletion policy means that each port an *IPRewriter* allocates cannot be used again for at least 30 seconds. An *IPRewriter* uses ports 1024 through 65,535, and thus can handle no more than 2,150 connections per second. To avoid this limit, the experiments described here use a configuration with 32 *IPRewriter* elements, each with its own IP address. A *HashDemux* spreads the flows from the internal hosts over the *IPRewriters* based on destination address. When a packet arrives from the outside world, an *IPClassifier* decides which *IPRewriter* to send it to, based on destination address.

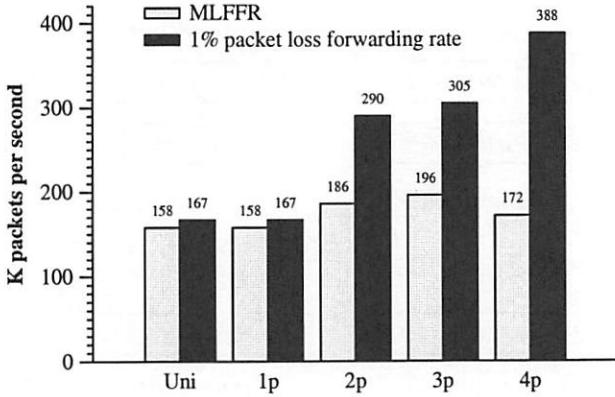


Figure 9: Performance of the NAT configuration under static CPU scheduling. The forwarding rate increases with more CPUs, but at the cost of some dropped packets.

The traffic used to test the NAT configuration in Figure 8 is generated as follows. Two hosts are considered internal hosts, and two are external hosts. Each internal host maintains 100 concurrent connections to each of the external hosts. For each connection, an internal host repeats the following: it chooses random port numbers, sends a SYN packet, 8 64-byte data packets, and a FIN packet, then starts over. The external host echoes each packet, exchanging fields as appropriate.

Figure 9 shows the NAT's packet forwarding performance as the number of CPUs increases. Static scheduling is used, and the assignments are the same as those in Section 5.2. The experiments run for 90 seconds, of which only the last 60 are included in the statistics; this allows time for the *IPRewriter* tables to fill up and for entries to start being deleted.

Figure 9 shows that the MLFFR of the NAT does not increase significantly with more processors, even though the NAT requires more CPU time than IP forwarding alone. The packet loss rate experienced by the NAT, however, remains tiny for input rates substantially greater than the MLFFR. For example, with 4 CPUs, the loss rate does not exceed 0.1% until the offered load is above 270,000 packets per second. We suspect the persistent tiny loss rate is caused by lock contention. Contention for locks may occur when a *PollDevice* attempts to push a packet through a rewriter while another *PollDevice* is pushing a packet through the same rewriter, but on a different CPU. With 32 rewriters, the possibility of contention is small. However, each contention is potentially costly: the cycles spent spin-waiting for the lock may delay scheduling of a *PollDevice* and cause a device's receive DMA queue to overflow.

5.5 Enforcing Quality of Service

Figure 10 shows a configuration that provides a simple quality of service guarantee. Inserted before the *ARPQuerier* element in the IP router in Figure 2, this configuration fragment clas-

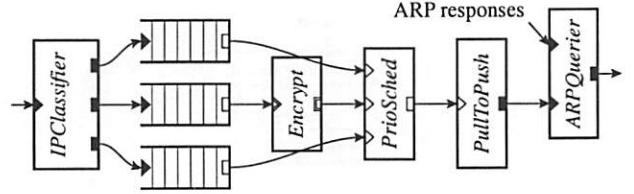


Figure 10: A QoS configuration fragment, to be inserted between the *IPFragmenter* and *ARPQuerier* elements in Figure 2.

sifies packets into three priorities based on the IP header's Differentiated Services Code Point field (DSCP) [22]. Each priority level is queued separately, and the *PrioSched* element always sends packets from higher priority queues in preference to lower. The configuration applies SHA-1 authentication and 3DES encryption to the medium priority traffic, much as described in Section 5.3. A *PullToPush* element initiates this processing, so it can be done on a CPU separately from input and output processing. The *PullToPush* element only pulls packets from upstream queues if the downstream queue is not full; this helps the configuration enforce priority when the output device is slow.

Figure 11 shows the uniprocessor performance of this configuration. The traffic used to test this router consists of three streams of UDP traffic from one host to another. The sender sends a high-priority and a low-priority stream at a constant 70,000 and 50,000 packets per second, respectively. It sends the third stream, of medium-priority traffic, with varying rate.

Figure 11 shows that as the input rate of medium priority traffic increases, the forwarding rate for high priority traffic does not change. There is enough spare CPU time that the medium priority traffic can be forwarded at up to 10,000 packets per second without disturbing the low priority traffic. Above that rate the router devotes CPU time to encrypt medium priority traffic at the expense of low-priority processing, so the low priority forwarding rate decreases. The specific mechanism is that the packet scheduling done by the *PrioSched* implicitly schedules the CPU, since the *PrioSched* decides which pull path the CPU executes. When the medium priority input rate reaches 14,000 packets per second, the forwarding rate levels off because all available CPU time has been taken from the low priority traffic.

Figure 12 shows that on four CPUs, the same IP router can sustain the low priority traffic even when the input rate of medium priority traffic approaches 17,000 packets per second. Furthermore, the maximum forwarding rate for medium priority traffic reaches 21,000 packets per second. The router uses dynamic scheduling. This causes three CPUs to handle device interactions and IP header processing; the remaining CPU runs the *PullToPush* element shown in Figure 10, and thus performs the VPN encryption as well as moving packets of all priorities through the configuration fragment.

These experiments show that SMP Click configurations

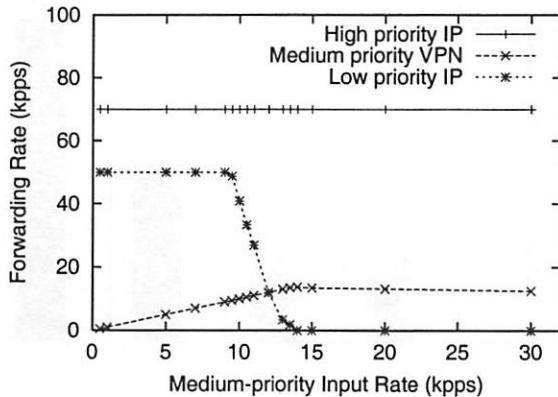


Figure 11: Performance of the QoS router on one CPU. The offered load consists of a constant 70 kpps high priority traffic, a constant 50 kpps low priority traffic, and an increasing rate of medium priority traffic. The y-axis shows the individual forwarding rates; the x-axis shows the medium-priority input rate.

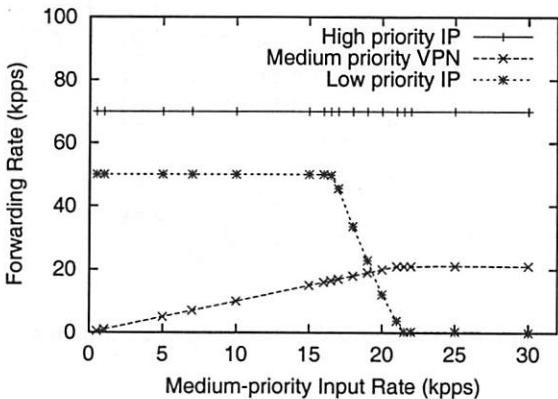


Figure 12: Performance of the QoS router on four CPUs. The extra CPUs improve performance over the uniprocessor results in Figure 11.

that express packet priority also naturally imply CPU priority. In addition, priority constraints do not prevent SMP Click from obtaining a degree of increased performance from multiple CPUs.

6 Exposing Parallelism

Even if a router's task has a good deal of potential parallelism, any given configuration may fail to expose that parallelism. Consider the VPN router in Section 5.3, but with only two interfaces instead of four. Such a configuration has four schedulable tasks. Most of the processing occurs in the push paths initiated by the two *PollDevice* tasks, so the CPUs running the *ToDevice* tasks may spend much of their time idle. A better

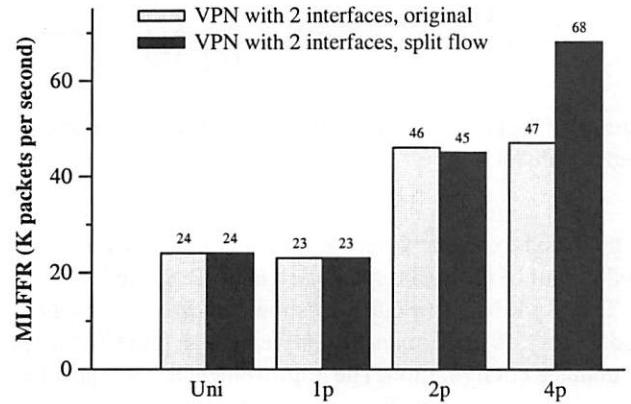


Figure 13: Performance of the VPN configuration with only two Ethernet devices. Processing separate flows in parallel increases the 4-CPU performance by 45%.

balance of load across the CPUs would increase performance, but the configuration doesn't allow for it.

This section presents and evaluates configuration tuning techniques that yield improvements in performance by exposing more parallelism. One technique splits packets into multiple flows which can be processed in parallel. A second technique breaks expensive processing into pipeline stages that can be executed in parallel. Finally, we show that configuration rewriting need not affect quality of service guarantees.

6.1 Parallel Flow Processing

Figure 13 shows the performance of the VPN configuration with only two Ethernet interfaces. Two hosts participate in these experiments. One host sends unencrypted packets to the router. The router encrypts the packets and forwards them onto the other host. The second host sends encrypted packets to the router. The router decrypts these packets and forwards them to the first host. Since encryption and decryption only occur on packets going to and arriving from one of the two devices, one *PollDevice* performs all the encryption work, and one *PollDevice* performs all the decryption work. This suggests that, on a four-CPU machine, the two CPUs running *ToDevices* are mostly idle. Consequently, the configuration doubles its performance on two processors, but its performance on four processors does not improve. More parallelism could be created by moving expensive elements to the pull paths, allowing *ToDevice* elements to share the expensive work. This turns out to be awkward; for example, at that point the packets already have Ethernet headers.

We create more parallelism by splitting packets into multiple flows. Two sets of *HashDemux*, *Queue*, and *PullToPush* elements are inserted before both the encryption and decryption elements in the VPN configuration, as suggested in Figure 3. This optimization creates two new schedulable elements: a *PullToPush* element that handles half of the pack-

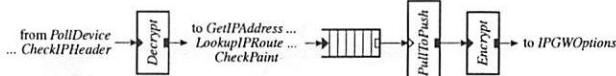


Figure 14: Pipelining allows decryption and encryption to proceed in parallel.

ets that need to be encrypted; and a *PullToPush* element that handles half of the packets that need to be decrypted.

The black bars in Figure 13 show that this optimization produces a 45% performance improvement on four CPUs over the untuned configuration. The improvement is not higher because of the cost of moving each packet from one CPU to another through the *Queue*.

This rewriting technique is not universal. For example, it decreases the performance of a router that does just IP forwarding between two interfaces. This is because the cost of vanilla IP forwarding is already low enough that parallelization cannot overcome the cost of the extra elements and cache misses.

6.2 Pipelined Packet Processing

Consider a node in an overlay network consisting of a mesh of encrypted VPN tunnels. Such a node may have to decrypt a packet arriving on one tunnel, only to encrypt it again (with a different key) when forwarding it out a second tunnel. Each *PollDevice* would be responsible for both decryption and encryption. On the other hand, the *ToDevices* would have relatively little work. More parallelism could be created by splitting packets into multiple flows, as described in Section 5.3. Parallelism can also be created by pipelining encryption and decryption, as shown in Figure 14. The technique is to insert a *Queue* and a *PullToPush* between the decryption and encryption processing. With this optimization, each *PollDevice* performs decryption in parallel with encryption performed by the *PullToPush*.

Figure 15 shows the effectiveness of this technique. It almost doubles the performance of the 4-CPU machine, and causes performance to scale almost linearly from one to four CPUs.

6.3 Maintaining Quality of Service

The performance improvement with additional CPUs of the QoS configuration described in Section 5.5 is limited, since a single CPU executes the *PullToPush* and thus the encryption. One solution might be to add a new *PullToPush* dedicated to the encryption of medium-priority traffic, leaving the old *PullToPush* to process only high and low priority traffic. Since a separate CPU could run the new *PullToPush*, performance should improve.

Rewriting the configuration this way, however, would violate the intended packet priority semantics. The old *PullTo-*

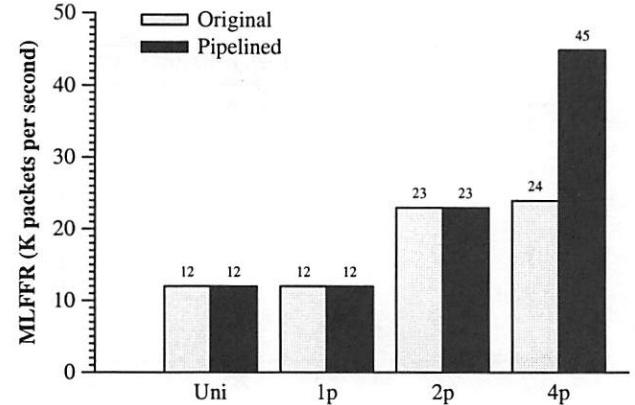


Figure 15: Performance of decryption and encryption with and without pipelining. Pipelining nearly doubles the 4-CPU performance.

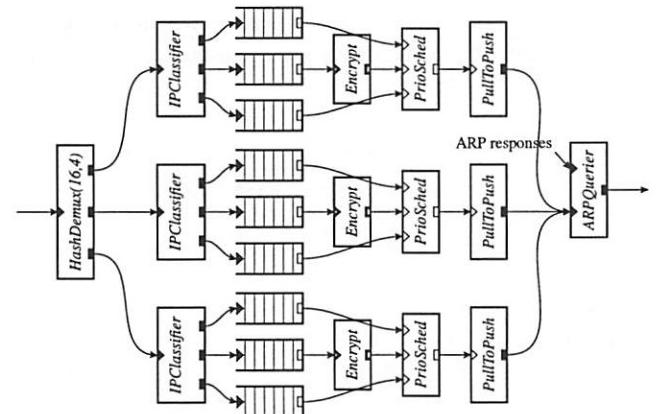


Figure 16: Splitting QoS processing into 3 streams.

Push would send low-priority packets even if there were a backlog of medium-priority packets, since only the new *PullToPush* would be able to process medium-priority packets. The intent of priority, however, is that low-priority packets should only be sent if there are no high or medium priority packets waiting.

A better approach is to replicate the whole QoS configuration and run the replicas in parallel. We replicate the configuration three times, as shown in Figure 16. The *HashDemux* element breaks packets into three streams; each stream has its own priority scheduler. The three *PullToPush* elements run on three CPUs, while the remaining CPU performs all the device handling and IP header processing. While it is possible that low priority packets are pushed through *ARPQuerier* on one processor while there are a backlog of medium or high priority packets on another, such scenario is unlikely when there are many flows with different destination IP addresses.

Figure 17 shows the effectiveness of this technique. The new router can sustain the low priority traffic even when the input rate of the medium priority VPN traffic exceeds 50,000

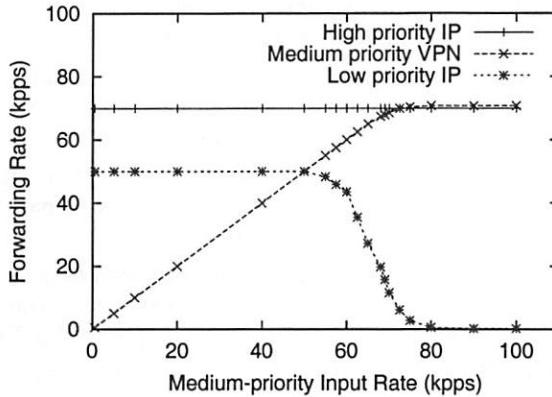


Figure 17: Performance of the QoS router on four CPUs with the new configuration in Figure 16. The performance is higher than that shown in Figure 12, but still honors the priority rules.

packets per second. The maximum forwarding rate for the medium priority traffic reaches 70,000 packets per second, a factor of five improvement over the uniprocessor performance shown in Figure 11.

7 Related Work

Commercial routers such as the Cisco 7500 [26] often contain multiple CPUs to increase performance. The hardware of such routers usually dedicates each CPU to a particular task. This structure provides high performance for its intended task, but allows little flexibility. For example, one line card's CPU cannot help in the processing of packets from a different line card. Commercial SMP routers do exist. The Nortel Contivity 4500 VPN switch [23] uses dual SMP PC processors to encrypt and decrypt packets for multiple VPN tunnels in parallel. Its hardware is similar to SMP Click's, though its software structure is not publically known.

In a different approach to multiprocessor routing, *network processors* [10, 6] have appeared recently that integrate multiple RISC CPUs onto a single chip. These chips could be placed on router line cards, replacing ASICs; their advantage is that it is easier and faster to write software than to design ASIC hardware. A variant of SMP Click could be used to structure that software in a way that takes advantage of the multiple CPUs. However, current generation network processors have a limited program memory in their processing elements, which limit their use to small pieces of tight code [25].

Previous work in the area of parallelizing host network protocols [20, 19, 3, 24] has compared layer, packet, and connection parallelism. One of their conclusions is that performance is best if the packets of each connection are processed on only one CPU, to avoid contention over per connection data. To a first approximation this is a claim that a host's protocol processing tasks can be decomposed into symmetric

and independent per-connection tasks. This situation does not generally hold in a router. If the router has no per-connection state, then there is no symmetry and independence to exploit. Worse, device handling is often a large fraction of the total work, but cannot easily be divided up among many CPUs. For these reasons, SMP Click needs to be able to exploit a wider range of kinds of parallelism than host implementations.

Blackwell [4] and Nahum et al. [18] investigate the interaction of host protocol processing and caching on uniprocessors. They observe that instruction cache misses are often a dominant factor in performance, and observe that batch processing of multiple packets at each protocol layer can help. In contrast, we observe very few instruction cache misses in SMP Click, probably because IP forwarding is simpler than host TCP processing. SMP Click nevertheless benefits from batching, though the reason is that batching helps avoid contention at the points where data must move between CPUs or between CPU and device.

SMP Click's device handling uses ideas explored in the Osiris [8] network adaptor project to maximize concurrency between CPU and device, in particular lock-free DMA queues and avoidance of programmed I/O.

8 Conclusion

This paper makes the following points about parallelization on multi-processor routers, in the context of SMP Click:

- Significant parallelism can often be found even in untuned configurations.
- Parallelization techniques can be effectively expressed at the level of router configurations, and such configurations can be restructured to enhance multiprocessor performance.
- Most cache misses in SMP Click occur when packets or buffer data structure move between CPUs or between CPU and device. This is in contrast to experience with host protocols, where instruction or protocol state misses dominate.
- Cache misses are expensive. Adaptive load-balancing the work on a multi-processor router may introduce more cache misses when packets move between CPUs. In most cases, a static scheduling assignment that minimizes the number of packets moving between CPUs can be found.
- Good multiprocessor routing performance requires concurrency in device interactions, both between CPUs and devices and between input and output on the same device.

- When packets need to move between CPUs, they should do so in batches to reduce per-packet contention overhead. Allocation and freeing of packet buffers is an important source of buffer data structure movement.

Availability

SMP Click can be downloaded from the Click project web page, at <http://www.pdos.lcs.mit.edu/click/>.

Acknowledgments

Eddie Kohler and Massimiliano Poletto helped us enormously with Click and *IPRewriter*. We also would like to thank Alex Snoeren for his work on the IPsec elements. We used Eric Young's DES and SHA-1 implementations. Michael Ernst, Frans Kaashoek, Rob Miller, and the anonymous reviewers provided us with invaluable feedback.

References

- [1] T. Anderson, E. Lazowska, and H. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.
- [2] F. Baker, editor. Requirements for IP Version 4 routers. RFC 1812, Internet Engineering Task Force, June 1995. <ftp://ftp.ietf.org/rfc/rfc1812.txt>.
- [3] M. Bjorkman and P. Gunningberg. Locking effects in multiprocessor implementation of protocols. In *Proc. ACM SIGCOMM '93 Conference*, pages 74–83, October 1993.
- [4] T. Blackwell. Speeding up protocols for small messages. In *Proc. ACM SIGCOMM '96 Conference*, pages 85–95, August 1996.
- [5] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th ACM Annual Symposium on Foundations of Computer Science*, pages 356–368, November 1994.
- [6] C-Port/Motorola. C-Port C-5 Network Processor. <http://www.cportcorp.com/products/digital.htm>.
- [7] Digital Equipment Corporation. DIGITAL Semiconductor 21140A PCI Fast Ethernet LAN Controller Hardware Reference Manual, March 1998. <http://developer.intel.com/design/network/manuals>.
- [8] P. Druschel, L. Peterson, and B. Davie. Experiences with a high-speed network adaptor: A software perspective. In *Proc. ACM SIGCOMM '94 Conference*, pages 2–13, August 1994.
- [9] K. Egevang and P. Francis. The IP network address translator (NAT). RFC 1631, Internet Engineering Task Force, May 1994. <ftp://ftp.ietf.org/rfc/rfc1631.txt>.
- [10] T. Halfhill. Intel network processor targets routers. *Microprocessor Report*, 13(12), September 1999.
- [11] Intel. IXP1200 Network Processor Datasheet. <http://www.intel.com/design/network/datashts/278298.htm>.
- [12] S. Kent and R. Atkinson. Security architecture for the internet protocol. RFC 2401, Internet Engineering Task Force, November 1998. <ftp://ftp.ietf.org/rfc/rfc2401.txt>.
- [13] Eddie Kohler. The Click modular router, PhD Thesis, MIT, 2000. <http://www.pdos.lcs.mit.edu/papers/click:kohler-phd/thesis.pdf>.
- [14] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(4):263–297, November 2000.
- [15] Eddie Kohler, Robert Morris, and Massimiliano Poletto. Modular components for network address translation. Technical report, MIT LCS Click Project, December 2000. <http://www.pdos.lcs.mit.edu/papers/click-rewriter/>.
- [16] M. McKusick, K. Bostic, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley Longman, 1996.
- [17] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. *ACM Trans. Computer Systems*, 15(3):217–252, August 1997.
- [18] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Cache behavior of network protocols. In *Proc. ACM SIGMETRICS '97 Conference*, pages 169–180, June 1997.
- [19] E. Nahum, D. Yates, S. O'Malley, O. Orman, and H. Schroeppel. Parallelized network security protocols. In *Proc. IEEE Symposium on Network and Distributed Systems*, pages 145–154, February 1996.
- [20] Erich M. Nahum, David J. Yates, James F. Kurose, and Don Towsley. Performance issues in parallelized network protocols. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 125–137, November 1994.
- [21] David Newman and Drew Olewnick. IPsec VPNs: How safe? how speedy? *The CommWeb Magazine Network*, September 2000. <http://www.commweb.com/article/COM20000912S0009>.
- [22] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services field (DS field) in the IPv4 and IPv6 headers. RFC 2474, Internet Engineering Task Force, December 1998. <ftp://ftp.ietf.org/rfc/rfc2474.txt>.
- [23] Nortel. Contivity VPN switches. <http://www.nortelnetworks.com/products/01/contivity/techspec.html>.
- [24] D. Schmidt and T. Suda. Measuring the performance of parallel message-based process architectures. In *Proc. IEEE Infocom '95*, pages 624–633, April 1995.
- [25] T. Spalink, S. Karlin, and L. Peterson. Evaluating network processors in IP forwarding. Technical report 626-00, Princeton University, November 2000. <http://www.cs.princeton.edu/nsg/router/papers/ixp.html>.
- [26] R. White, V. BollaPragada, and C. Murphy. *Inside Cisco IOS Software Architecture*. Cisco Press, 2000.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *:login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Java, Tcl/Tk, and Open Source, book and software reviews, summaries of sessions at USENIX conferences, and Standards Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to *:login:* on the USENIX Web site.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, on the USENIX Web site.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as security, Linux, Internet technologies and systems, operating systems, and Windows—as many as twelve technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- The right to vote on matters affecting the Association, its bylaws, and election of its directors and officers.
- Savings on a variety of products, books, software, and periodicals: see <http://www.usenix.org/membership/specialdisc.html> for details.

Supporting Members of the USENIX Association:

Addison-Wesley	Microsoft Research	Sendmail, Inc.
Kit Cosper	Motorola Australia Software Centre	Smart Storage, Inc.
Earthlink Network	New Riders Publishing	Sun Microsystems, Inc.
Edgix	Nimrod AS	Sybase, Inc.
Interhack Corporation	O'Reilly & Associates Inc.	Syntax, Inc.
Interliant	Raytheon Company	Taos: The Sys Admin Company
Lessing & Partner	Sams Publishing	TechTarget.com
Linux Security, Inc.	The SANS Institute	UUNET Technologies, Inc.
Lucent Technologies		

Supporting Members of SAGE:

Certainty Solutions	Mentor Graphics Corp.	Remedy Corporation
Collective Technologies	Microsoft Research	RIPE NCC
Electric Lightwave, Inc.	Motorola Australia Software Centre	Sams Publishing
ESM Services, Inc.	New Riders Publishing	SysAdmin Magazine
Lessing & Partner	O'Reilly & Associates Inc.	Taos: The Sys Admin Company
Linux Security, Inc.	Raytheon Company	Unix Guru Universe

For more information about membership, conferences, or publications,

see <http://www.usenix.org/>

or contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 Fax: 510-548-5738 Email: office@usenix.org

ISBN 1-880446-09-X